PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

A1

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 12/00 | A1

(11) International Publication Number: WO 96/12227

(43) International Publication Date: 25 April 1996 (25.04.96)

(21) International Application Number: PCT/US95/13299

(22) International Filing Date: 13 October 1995 (13.10.95)

(30) Priority Data:
08/324,129 14 October 1994 (14.10.94) US

(71) Applicant: SILICON GRAPHICS, INC. [US/US]; 2011 North Shoreline Boulevard, Mountain View, CA 94043-1389 (US).

(72) Inventor: YEAGER, Kenneth, C.; Apartment 15-205, 1063 Morse, Sunnyvale, CA 94089 (US).

(74) Agents: KESSLER, Edward, J. et al.; Sterne, Kessler, Goldstein & Fox, P.L.L.C., Suite 600, 1100 New York Avenue, N.W., Washington, DC 20005-3934 (US).

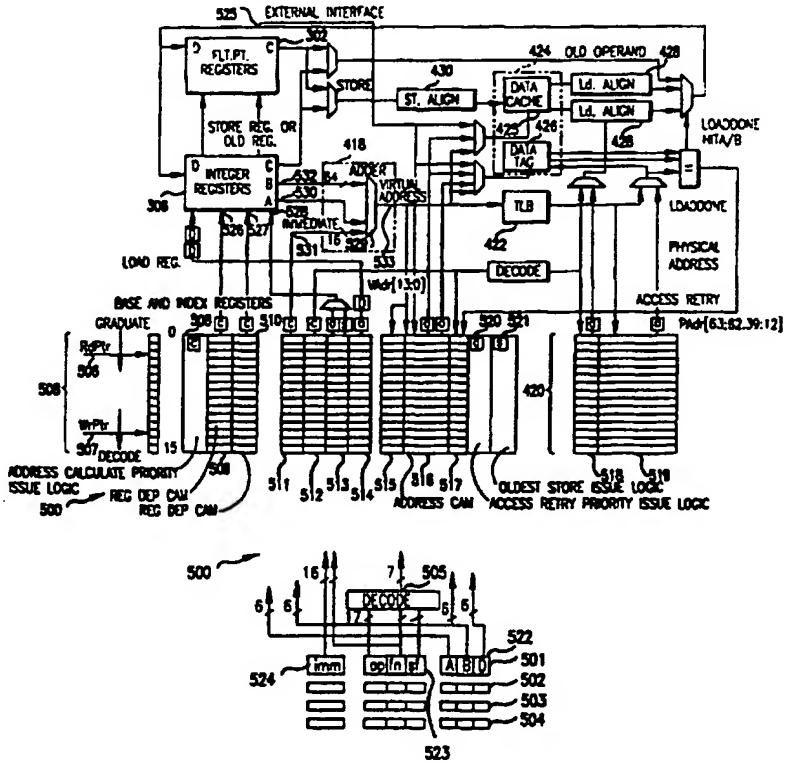
(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: AN ADDRESS QUEUE CAPABLE OF TRACKING MEMORY DEPENDENCIES

(57) Abstract

An address queue (308) in a superscalar processor has the capability to track memory dependency of memory access instructions that may be executed out-of-order. In accessing a two way-set-associative data cache, the address queue imposes a dependency for accesses to the same cache set to prevent unnecessary cache trashing. The address queue (308) holds a plurality of entries used to access a set-associative data cache. This queue includes a comparator circuit (2406), a first matrix (2400) of RAM cells and a second matrix (2450) of RAM cells. The comparator circuit (2406) compares a newly calculated partial address derived from a new queue entry with a previously calculated partial address derived from one of a number of previous entries. The first matrix (2400) of RAM cells tracks all of the previous entries in the queue that use a cache set that is also used by the new queue entry. The second matrix (2450) of RAM cells tracks queue entries that are store instructions which store a portion of data in the data cache which is accessed by a subsequent load instruction.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**GB**	United Kingdom	**MR**	Mauritania
AU	Australia	**GE**	Georgia	**MW**	Malawi
BB	Barbados	**GN**	Guinea	**NE**	Niger
BE	Belgium	**GR**	Greece	**NL**	Netherlands
BF	Burkina Faso	**HU**	Hungary	**NO**	Norway
BG	Bulgaria	**IE**	Ireland	**NZ**	New Zealand
BJ	Benin	**IT**	Italy	**PL**	Poland
BR	Brazil	**JP**	Japan	**PT**	Portugal
BY	Belarus	**KE**	Kenya	**RO**	Romania
CA	Canada	**KG**	Kyrgyzstan	**RU**	Russian Federation
CF	Central African Republic	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CG	Congo			**SE**	Sweden
CH	Switzerland	**KR**	Republic of Korea	**SI**	Slovenia
CI	Côte d'Ivoire	**KZ**	Kazakhstan	**SK**	Slovakia
CM	Cameroon	**LI**	Liechtenstein	**SN**	Senegal
CN	China	**LK**	Sri Lanka	**TD**	Chad
CS	Czechoslovakia	**LU**	Luxembourg	**TG**	Togo
CZ	Czech Republic	**LV**	Latvia	**TJ**	Tajikistan
DE	Germany	**MC**	Monaco	**TT**	Trinidad and Tobago
DK	Denmark	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**US**	United States of America
FI	Finland	**ML**	Mali	**UZ**	Uzbekistan
FR	France	**MN**	Mongolia	**VN**	Viet Nam
GA	Gabon				

AN ADDRESS QUEUE CAPABLE OF TRACKING MEMORY DEPENDENCIES

Background of the Invention

This invention relates in general to computers capable of executing instructions out of order and, in particular, to a computer capable of tracking dependencies between out-of-order instructions that are used to access memory.

From the perspective of a programmer, instructions in a conventional processor are executed sequentially. When an instruction loads a new value into its destination register, that new value is immediately available for use by subsequent instructions. This is not true, however, for pipelined computer hardware because some results are not available for many clock cycles. Sequencing becomes more complicated in a superscalar processor, which has multiple execution pipelines running in parallel. But the hardware must behave as if each instruction were completed sequentially.

Each instruction depends on previous instructions which produced its operands, because it cannot begin execution until those operands become valid. These dependencies determine the order in which instructions can be executed. The actual execution order depends on the organization of the processor. In a typical pipelined processor, instructions are executed only in program order. The next sequential instruction may begin execution during the next cycle provided all its operands are valid. Otherwise, the pipeline stalls until the operands become valid. Because instructions execute in order, stalls usually delay all subsequent instructions. A sophisticated compiler can improve performance by re-arranging instructions to reduce the frequency of these stall cycles.

In an in-order superscalar processor, several consecutive instructions may begin execution simultaneously, if all their operands are valid, but the processor stalls at any instruction whose operands are still busy. In an out-of-order superscalar processor, each instruction is eligible to begin execution as soon as its operands become valid, independently of the original instruction sequence. In effect, the hardware re-arranges instructions to keep its execution units busy. This process is called "dynamic issuing."

Dynamic issue and execution of pipelined instructions creates a special need to monitor and resolve data dependencies between instructions. A newly-issued instruction is dependent on a previous instruction if, for example, the newly-issued instruction must use an output of the previous instruction as an operand. Such dependency inserts a restriction on the order of instruction execution.

Similarly, when out-of-order instructions are used in memory-access operations, the execution order of such instructions is restricted, at least in part, by memory dependency (i.e., two instructions accessing and altering the same memory location). Accordingly, there is a need for tracking the memory-dependency of memory-access instructions which may be executed out of order to maintain data integrity.

Summary of the Invention

The present invention offers a highly efficient apparatus for tracking memory dependencies of memory-access instructions that may be executed out of order. This apparatus also provides for special identification of portions of a memory cache set to prevent unnecessary cache thrashing.

In one embodiment, the present invention provides an address queue for holding a plurality of entries used to access a set-associative data cache. This queue includes a comparator circuit, first matrix of RAM cells and second matrix of RAM cells. The comparator circuit compares a newly calculated partial address derived from a new queue entry with a previously calculated partial address derived from one of a number of previous entries. The first matrix of RAM cells tracks all of the previous entries in the queue that use a cache set that is also used by the new queue entry. The second matrix of RAM cells tracks queue entries that are store instructions which store a portion of data in the data cache which is accessed by a subsequent load instruction. The address queue

may also assign status bits to certain blocks stored in the cache to identify the type of access allowed; i.e., random or sequential.

A better understanding of the nature and advantages of the present invention may be had with reference to the detailed description and the drawings below.

Brief Description of the Figures

Fig. 1 discloses a functional block diagram of a superscalar processor;

Fig. 2 discloses rows in the address queue disclosed herein;

Fig. 3 discloses physical layout of address queue;

Fig. 4 illustrates format of improved branch instruction formats;

Fig. 5 illustrates instruction set format;

Fig. 6 illustrates new instruction formats in Mips-4 ISA;

Fig. 7 discloses connection of the address queue;

Fig. 8 discloses instruction fields sent to the address queue and address stack;

Fig. 9 discloses address queue and address stack contents;

Fig. 10 defines the 7-bit operation codes stored in the address queue;

Fig. 11 is an address calculation timing diagram;

Fig. 12 illustrates priority protocol for using the data cache;

Fig. 13 discloses active-signal logic of the address queue;

Fig. 14 discloses generating an active mask;

Fig. 15 discloses priority logic of the address queue;

Fig. 16 is an example of retry access priority;

Fig. 17 discloses retry access priority logic of the address queue;

Fig. 18 discloses synchronize mask logic of the address queue;

Fig. 19 is an example of a synchronize mask;

Fig. 20 illustrates a high group within the synchronize mask;

Fig. 21 discloses access request logic of the address queue;

Fig. 22 discloses dependency comparators of the address queue;

Fig. 23 is an address queue timing diagram;

Fig. 24 discloses dependency matrix logic in the address queue;

Fig. 25 discloses dependency matrix operation in the address queue;

Fig. 26 discloses dependency checks during tag check cycles;

Fig. 27 discloses dependency comparator logic of the address queue;

Fig. 28 discloses dependency comparator circuits of the address queue;

Fig. 29 discloses byte overlap circuit of the address queue;

Fig. 30 discloses dependency matrix logic cells of the address queue; and

Fig. 31 is a dependency timing diagram.

Detailed Description of the Preferred Embodiment

Contents

I. Superscalar Processor Architecture

Fig. 1 discloses a functional block diagram of a superscalar processor 100 which incorporates an address queue built and operating in accordance with the present invention. As discussed below, this address queue enables, among other things, the tracking of memory dependencies in a processor that provides for out-of-order execution of instructions. Processor 100, which generally represents the R10000 Super-Scalar Processor developed by Silicon Graphics, Inc., of Mountain View, California, provides only one example of an application for the address queue of the present invention. This processor is described in J. Heinrich, *MIPS R10000 Microprocessor User's Manual*, MIPS Technologies, Inc., (1994).

A. Superscalar Processor Overview

A superscalar processor can fetch and execute more than one instruction in parallel. Processor 100 fetches and decodes four instructions per cycle. Each decoded instruction is appended to one of three instruction queues. These queues can issue one new instruction per cycle to each of five execution pipelines.

The block diagram of Fig. 1 is arranged to show the stages of an instruction pipeline and illustrates functional interconnectivity between various processor elements. Generally, instruction fetch and decode are carried out in stages 1 and 2; instructions are issued from various queues in stage 3; and instruction execution is performed in stages 4-7.

Referring to Fig. 1, a primary instruction cache 102 reads four consecutive instructions per cycle, beginning on any word boundary within a cache block. A branch target cache 104, instruction register 106 and instruction decode and dependency logic 200, convey portions of issued instructions to floating point mapping table 204 (32 word by 6 bit RAM) or integer mapping table 206 (33 word by 6 bit RAM). These tables carry out a "register renaming" operation, which renames logical registers identified in an instruction with a physical

register location for holding values during instruction execution. A redundant mapping mechanism is built into these tables to facilitate efficient recovery from branch mispredictions.

The redundant mapping tables are for use in processors that rename registers and perform branch prediction is presented. The redundant mapping tables include a plurality of primary RAM cells coupled to a plurality of redundant RAM cells. In the event of a branch instruction, the redundant RAM cells can save the contents of the primary RAM cells in a single clock cycle before the processor decodes and executes subsequent instructions along a predicted branch path. Should the branch instruction be mispredicted, the redundant cells can restore the primary RAM cells in a single clock cycle. A plurality of levels of redundant RAM cells may be used to enable the nesting of a plurality of branch predictions at any one time.

More particularly, the redundant mapping tables represent a highly efficient mechanism for saving and restoring register-renaming information to facilitate branch prediction and speculative execution. This mechanism enables the contents of register renaming mapping tables (i.e., memories used to rename logical register numbers to physical register numbers) to be saved and restored in a single clock cycle.

A register-renaming mapping table has a plurality of primary RAM cells. A plurality of redundant RAM cells are coupled to these primary cells and capable of saving and restoring the contents of the primary RAM cells in a single clock cycle. Multiple levels of redundant RAM cells may be employed to save multiple versions of the contents of the primary RAM cells.

Mapping tables 204 and 206 also receive input from a floating point free list 208 (32 word by 6 bit RAM) and an integer free list 210 (32 word by 6 bit RAM), respectively. Output of both mapping tables is fed to active list 212 which, in turn, feeds the inputs of free lists 208 and 210.

A branch unit 214 also receives information from instruction register 106, as shown in Fig. 1. This unit processes no more than one branch per cycle. The

branch unit includes a branch stack 216 which contains one entry for each conditional branch. Processor 100 can execute a conditional branch speculatively by predicting the most likely path and decoding instructions along that path. The prediction is verified when the condition becomes known. If the correct path was taken, processing continues along that path. Otherwise, the decision must be reversed, all speculatively decoded instructions must be aborted, and the program counter and mapping hardware must be restored.

Referring again to Fig. 1, mapping tables 204 and 206 support three general pipelines, which incorporate five execution units. A floating-point pipeline is coupled to floating-point mapping table 204. The floating-point pipeline includes a sixteen-entry instruction queue 300 which communicates with a sixty-four-location floating point register file 302. Register file 302 and instruction queue 300 feed parallel multiply unit 400 and adder 404 (which performs, among other things, comparison operations to confirm floating-point branch predictions). Multiply unit 400 also provides input to a divide unit 408 and square root unit 410.

Second, an integer pipeline is coupled to integer mapping table 206. The integer pipeline includes a sixteen-entry integer instruction queue 304 which communicates with a sixty-four-location integer register file 306. Register file 306 and instruction queue 304 feed two arithmetic logic units ("ALU"): ALU#1 412 (which contains an ALU, shifter and integer branch comparator) and ALU#2 414 (which contains an ALU, integer multiplier and divider).

Third, a load/store pipeline (or load/store unit) 416 is coupled to integer mapping table 206. This pipeline includes a sixteen-entry address queue 308 which communicates with register file 306. Address queue 308 is built and operates in accordance with the present invention.

Register file 306 and address queue 308 feed integer address calculate unit 418 which, in turn, provides virtual-address index entries for address stack 420. These virtual addresses are converted to physical addresses in translation

lookaside buffer (TLB) 422, and used to access a data cache 424 that holds data 425 and tags 426. These physical addresses are also stored in address stack 420.

The JTLB 422 is a translation buffer for detecting and preventing conflicting virtual addresses from being stored therein. The JTLB 422 is a highly efficient mechanism for implementing translations of virtual memory addresses into physical memory. The JTLB 422 accurately prevents identical virtual page addresses from being stored in the translation buffer without having to shut down and reset the translation buffer. The JTLB 422 includes a first register for storing a first physical page value, a second register for storing a second physical page value, a third register for storing the value representing the page size, a fourth register for storing a virtual address corresponding to the values stored in the first and second registers, and a comparator for comparing a current virtual address with the value in the third register. The comparator generates a signal if the compared values are equal. This signal causes either the first or second register to be read.

Data input to and output from data cache 424 pass through store aligner 430 and load aligner 428, respectively.

The data cache 424 is a memory array for storing instructions and data which is used by the CPU. Employing interleaved techniques, a high hit rate is achieved to effectively reduce the number of accesses to slower main memory. Multiplexing circuits enable the memory array to be more densely packed and implemented with lower numbers of sense amplifiers.

The cache 424 preferably operates in parallel with the translation lookaside buffer to reduce its latency. The cache 424 contains two 2-way set-associative arrays that are interleaved together. Each 2-way set-associative array includes two arrays, one each for the tag and data. By having four independently operating cache arrays, up to four instructions can operate simultaneously. The bits in each data array are interleaved to allow two distinct access patterns. For example, when the cache 424 is loaded or copied back, two double words in the same block are accessed simultaneously. When the cache 424 is read, the same

doubleword location is simultaneously read from both blocks with the set. Further, by using a multiplexer, the number of sense amplifiers for reading and writing are reduced, thereby saving significantly valuable space on the die.

Address stack 420 and data cache 424 also communicate with secondary cache controller and external interface 434. Further, data cache 424 and controller-interface 434 communicate with secondary cache 432. External interface 434 sends a 4-bit command (DCmd[3:0]) to data cache 424 and address queue 308 (connection not shown) to indicate what operation the cache will perform for it. Address queue 308 derives signals which control reading and writing from data cache 424.

B. Operation

Processor 100 uses multiple execution pipelines to overlap instruction execution in five functional units. As described above, these units include the two integer ALUs 412, 414, load/store unit 416, floating-point adder 404 and floating-point multiplier 400. Each associated pipeline includes stages for issuing instructions, reading register operands, executing instructions, and storing results. There are also three "iterative" units (i.e., ALU#2 414, floating-point divide unit 408 and floating-point square root unit 410) which compute more complex results.

Register files 302 and 306 must have multiple read and write ports to keep the functional units of processor 100 busy. Integer register file 306 has seven read and three write ports; floating-point register file 302 has five read and three write ports. The integer and floating-point execution units each use two dedicated operand ports and one dedicated result port in the appropriate register file. Load/store unit 416 uses two dedicated integer operand ports for address calculation. It must also load or store either integer or floating-point values, sharing a result port and a read port in both register files. These shared ports are

also used to move data between the integer and floating-point register files, and for "Jump and Link" and "Jump Register" instructions.

In a pipeline, the execution of each instruction is divided into a sequence of simpler operations. Each operation is performed by a separate hardware section called a stage. Each stage passes its result to the next stage. Usually, each instruction requires only a single cycle in each stage, and each stage can begin a new instruction while previous instructions are being completed by later stages. Thus, a new instruction can often begin during every cycle.

Pipelines greatly improve the rate at which instructions can be executed. However, the efficient use of a pipeline requires that several instructions be executed in parallel. The result of each instruction is not available for several cycles after that instruction entered the pipeline. Thus, new instructions must not depend on the results of instructions which are still in the pipeline.

Processor 100 *fetches* and *decodes* instructions in their original program order, but may *execute* and *complete* these instructions out of order. Once completed, instructions are "*graduated*" in their original program order. Instruction fetching is carried out by reading instructions from instruction cache 102, shown in Fig. 1. Instruction decode operation includes dependency checks and register renaming, performed by instruction decode and dependency logic 200 and mapping tables 204 or 206, respectively. The execution units identified above compute an arithmetic result from the operands of an instruction. Execution is complete when a result has been computed and stored in a temporary register identified by register file 302 or 306. Finally, graduation commits this temporary result as a new permanent value.

An instruction can graduate only after it and all previous instructions have been successfully completed. Until an instruction has graduated, it can be aborted, and all previous register and memory values can be restored to a precise state following any exception. This state is restored by "unnaming" the temporary physical registers assigned to subsequent instructions. Registers are unnamed by writing an old destination register into the associated mapping table

and returning a new destination register to the free list. Renaming is done in reverse program order, in the event a logical register was used more than once. After renaming, register files 302 and 306 contain only the permanent values which were created by instructions prior to the exception. Once an instruction has graduated, however, all previous values are lost.

Active list 212 is a list of "active" instructions in program order. It records status, such as which instructions have been completed or have detected exceptions. Instructions are appended to its bottom when they are decoded. Completed instructions are removed from its top when they graduate.

C. Instruction Queues

Processor 100 keeps decoded instructions in three instruction queues. These queues dynamically issue instructions to the execution units. Referring to Fig. 2, the entries in each queue are logically arranged in four rows (i.e., rows 220-226) of four entries 218, as shown in Fig. 2. (This "row" and "column" terminology is figurative only; the physical layout has one column of sixteen entries 350, as shown in Fig. 3.) While an instruction queue has four write ports 236-242, each queue entry 218 has only a single write port 219. Entries within each row share the same queue write port, but each row has a separate port. These inputs are fed from four four-to-one multiplexers 228-234 (MUXes) which can select any of the four instructions currently being decoded. These MUXes align new instructions with an empty entry. A new instruction can be written into each row if it has at least one empty entry.

1. Integer Queue

Integer queue 304, as shown in Fig. 1, issues instructions to two integer arithmetic units: ALU#1 412 and ALU#2 414. This queue contains 16 instruction entries. Newly decoded integer instructions are written into empty entries

without any particular order. Up to four instructions may be written during each cycle. Instructions remain in this queue only until they have been issued to an ALU. Branch and shift instructions can be issued only to ALU#1 412. Integer multiply and divide instructions can be issued only to ALU#2 414. Other integer instructions can be issued to either ALU.

The Integer Queue controls six dedicated ports to integer register file 306. These include two operand read ports and a destination write port for each ALU.

2. Floating Point Queue

Floating-point queue 300, as shown in Fig. 1, issues instructions to floating-point multiplier 400 and floating-point adder 404. This queue contains 16 instruction entries. Newly decoded floating-point instructions are written into empty entries without any particular order. Up to four instructions may be written during each cycle. Instructions remain in this queue only until they have been issued to a floating-point execution unit.

The Floating-point queue controls six dedicated ports to floating-point register file 302. These include two operand read ports and a destination port for each execution unit. Queue 300 uses the issue port of multiplier 400 to issue instructions to square-root unit 410 and divide unit 408. These instructions also share the register ports of multiplier 400.

Further, Floating-Point queue 300 contains simple sequencing logic for multi-pass instructions, such as Multiply-Add. These instructions require one pass through multiplier 400 and then one pass through the adder 404.

3. Address Queue

Address queue 308, as shown in Fig. 1, issues instructions within load/store unit 416. It contains 16 instruction entries. Unlike the other two queues, address queue 308 is organized as a circular First-In-First-Out ("FIFO")

buffer. Newly decoded load/store instructions are written into the next sequential empty entries. Up to four instructions may be written during each cycle. The FIFO order maintains the program's original instruction sequence so that memory address dependencies may be computed easily. Instructions remain in this queue until they have graduated. They cannot be deleted immediately after being issued, because load/store unit 416 may not be able to immediately complete the operation.

Address queue 308 contains more complex control logic than the other queues. An issued instruction may fail to complete, because of a memory dependency, a cache miss, or a resource conflict. In these cases, the queue must re-issue the instruction until it has graduated.

Address queue 308 has three issue ports; issue, access and store. The first two are dynamic (i.e., may issue instructions out of order) while the third issues instructions in order. First, address queue 308 issues each instruction once to address calculation unit 418. This unit uses a 2-stage pipeline to compute the memory address of an instruction and translate it in Translation Look-aside Buffer 422 ("TLB"). Addresses are stored in address stack 420 and in the dependency logic of the queue, as discussed below. This port controls two dedicated read ports to integer register file 306. This logic is similar to the other queues. Issue port may use tag check circuitry (discussed below) if it is not used by the access port.

Second, address queue 308 can issue "accesses" to data cache 424. The queue allocates usage of four sections of the cache, which consist of tag and data sections of the two cache banks. Load and store instructions begin with a tag check cycle, which checks if the desired address is already in cache. Load instructions also read and align a doubleword value from the data array. This access may be concurrent or later than the tag check. If the data is present and no dependencies exist, the instruction is marked "done" in the queue.

Third, address queue 308 can issue "store" instructions to the data cache. A store instruction may not modify the data cache until it graduates. Only one

store can graduate per cycle, but it may be anywhere within the four oldest instructions, if all previous instructions are already "done".

The "Access" and "Store" ports share two integer and two floating-point register file ports. These "shared" ports are also used for "Jump and Link" and "Jump Register" instructions and for move instructions between the integer and register files.

D. Coprocessors

Processor 100 can operate with up to four tightly-coupled coprocessors (designated CP0 through CP3). Coprocessor unit number zero (CP0) supports the virtual memory system together with exception handling. Coprocessor unit number one CP1 (and unit three (CP3) in Mips-4 Instruction Set Architecture, discussed below) is reserve for floating-point operations.

E. Instruction Formats

Processor 100 implements the Mips-4 Instruction Set Architecture ("ISA") and is compatible with earlier Mips-1, Mips-2 and Mips-3 ISAs. The formats of these instructions are summarized in Figs. 4-6. Fig. 4 shows jump and branch instruction formats. The "Branch on Floating-Point Condition" (CP1) instruction was enhanced in Mips-4 ISA to include eight condition code bits, instead of the single bit in the original instruction set. In Mips-3 or earlier ISA, the three-bit condition code select field ("CC") must be zero. Fig. 5 shows other formats in the Mips-3 and earlier ISAs. A discussion of Mips ISAs is provided in J. Heinrich, *MIPS R4000 User's Manual*, PTR Prentice Hall (1993) and G. Kane *et al.*, *MIPS RISC Architecture*, Prentice Hall (1992). A description of Mips-4 ISA is provided in C. Price, *MIPS R10000 - Mips IV ISA Manual*, MIPS Technologies, Inc. (1994).

The extensions for the MIPS-4 ISA are shown in Fig. 6. These new instructions facilitate floating-point and vector programs. These include floating-point multiply-add, double-indexed load and store floating-point, and conditional move instructions. The floating-point compare instruction has been modified to set any of eight condition bits.

II. Address Queue

Address Queue 308 keeps track of all memory instructions in the pipeline. As noted above, it contains 16 entries, which are organized as a circular FIFO buffer or list 500, as indicated in Fig. 7. When a memory load or store instruction is decoded, it is allocated to the next sequential entry at the "bottom" of list 500, which includes list segments 509-514. Any or all of the four instructions (i.e., 501-504) decoded during a cycle may be loads or stores, so up to four instructions may be appended to address queue 308 in one cycle.

Fig. 7 shows the loading of an instruction containing portions 522-524 to list 500. Portion 522 contains mapped physical register numbers for operands A and B and destination D, which are appended to "A" segment 509, "B" segment 510 and "D" segment 514, respectively. Operand B is also appended to "C" segment 513 (see Table 1). Further, instruction portion 523 contains opcode ("op"), function ("fn") and subfunction ("sf") values which are decoded in decode logic 505 and appended to "function" segment 512. Finally, instruction portion 522 contains an "immediate" value (described below) which is appended to "immediate" segment 511 along with a portion of the "fn" value from portion 523.

Remaining portions of list 500 include "dependency" segment 515, "index" segment 516 and "state" segment 517. Also shown is address stack 420 which includes "control" segment 518 and "physical address" segment 519. Blocks 508, 520 and 521 represent issue logic which is described below.

The instruction fields sent to the address queue and address stack are summarized in Fig. 8.

Instructions are deleted from the "top" of list 500 when they graduate. Up to four instructions can graduate per cycle, but only one of these can be a "store". Instructions may also be deleted from the bottom when a speculative branch is reversed and all instructions following the branch are aborted. The queue is cleared when an exception is taken. Various address queue contents are summarized in Fig. 9.

The FIFO uses two 5-bit pointers 506 and 507, as shown in Fig. 7. The low four bits select one of the 16 entries. The high bit detects when the bottom of the queue has "wrapped" back to the top. "Write" pointer 507 selects the next empty entry. "Read" pointer 506 selects the oldest entry, which will be the next to graduate. The write pointer is copied onto branch stack 216 (Fig. 1) in one of four "shadow" pointers whenever a speculative branch is taken. It is restored from the corresponding shadow if the branch decision is reversed.

Because of layout restrictions, address queue 308 is physically implemented in two sections. The "address queue" section (i.e., segments and blocks 508-517, 520 and 521), which is located between instruction decode 200 and integer register file 306, contains most of the control logic. It contains the base and index register fields, and address offset field, and it issues instructions to address calculation unit 418. It also issues load and store instructions to data cache 424 and resolves memory dependencies. The address stack section (i.e., segments 518, 519), which is located near TLB 422, contains the translated physical address.

The Address Queue has three issue ports. The "Calculate" port issues each instruction once to address calculation unit 418, TLB 422, and (if available) to data cache 424. The "Access" port is used to retry instructions. The "Store" port is used to graduate store instructions.

A. Contents

Address queue 308 contains "Load" and "Store" instructions, which access data cache 424 and main memory. It also contains "Cache" instructions, which manipulate any of the caches. This queue also controls address calculation circuit 418. For "Indexed" instructions, it provides two physical register operands via integer register file 306. For other load or store instructions, it provides one register operand and a 16-bit immediate value.

The fields within each address queue entry are listed in tables 1-3. Table 1 lists instruction fields, which contain bits that are loaded into segments 509-514 (Fig. 7) after an instruction is decoded.

Field (AQv-___)	Description
ActiveF	Indicates entry is active. This signal is decoded from queue pointers 506 and 507, and then delayed one cycle in a register. This signal enables stack retry requests. (1 bit.)
Tag	Active List tag uniquely identifies an instruction within the pipeline. (5 bits.)
Func	Instruction opcode and function. Address queue 308 receives a 17-bit function code from decode logic 200. It condenses this to a 7-bit code.
0nnnnnn	6-bit major opcode (modified during instruction predecode), or
10nnnnn	6-bit function code from COP1X opcode. (AQ gets codes #00-#37 octal only.)
11 fff cc	5-bit subfunction code for CACHE operations (3-bit function, 2-bit cache select.)
Imm	The immediate field contains the address offset from instruction bits. During decode, the high 10 bits are from instruction portion 524 and the low 6 bits are from portion 523 (Fig. 7). (16 bits.)
	<u>Base Register</u>:
OpSelA	Operand A, select physical register # in integer register file 306. (6 bits.)
OpRdyA	Operand A is ready for address calculation. (1 bit.)
OpValA	Operand A is valid for address calculation. (Integer register # is not zero; 1 bit.)

Field (AQv-___)	Description
OpSelB	Index Register or Integer Operand: Operand B, select physical register # in integer register file 306. (For integer stores, this value is duplicated in AQvOpSelC; 6 bits.)
OpRdyB	Operand B is ready. (1 bit.)
OpValB	Operand B is valid. (Integer register # is not zero; 1 bit.)
OpSelC	Floating-point Operand: Operand C, select physical register # in flt. pt. register file 302. (For integer stores, this field contains a copy of AQvOpSelB; 6 bits.)
OpRdyC	Operand C is ready. (1 bit.)
OpValC	Operand C is valid. (1 bit.)
Dest	Destination, select physical register #. (6 bits.)
DType	Destination type (or hint; 2 bits): 00 = No destination register. (If prefetch instruction, hint = "shared".) 01 = No destination register. (If prefetch instruction, hint = "exclusive".) 10 = Integer destination register. 11 = Floating-point destination register.
UseR	Which ports of the shared register files are required to execute this instruction (4 bits)? Bit 3: Flt.pt. Write. Bit 2: Flt.pt. Read. Bit 1: Integer Write. Bit 0: Integer Read.
Store	This instruction is a store. (1 bit.)
Flt	This instruction loads or stores a floating-point register. (1 bit.)
FltHi	Load or store high half of floating-point register (if FR=0; 1 bit).

Table 1: Address Queue Instruction Fields

With respect to the AQvFunc entry listed in Table 1, a "predecode" operation partially decodes an instruction as it is written into instruction cache 102 during a refill operation (i.e., refilling cache 102 in the event of a miss). This step re-arranges fields within each instruction to facilitate later decoding. In

particular, the register select fields are arranged for convenient mapping and dependency checking. The destination and source register fields are put into fixed locations, so they can be used directly as inputs to mapping tables 204 and 206 (Fig. 1), without further decoding or multiplexing.

Table 2 below lists address and dependency bits, which are loaded from address calculation unit 418 and held in segments 515 and 516, respectively, as shown in Fig. 7. The dependency bits are updated continuously as previous instructions graduate and other instructions are calculated.

Field (AQv-__)	Description
Index [13:3]	Primary cache index address. Bits [13:5] select a set within the primary data cache. Set contains two 8-word blocks. Bit [5] selects either Bank 0 (if 0) or Bank 1 (if 1) of the primary data cache. Bit [4:3] select a doubleword within an 8-word cache block. (Bits [2:0] are decoded into an 8-bit byte mask, which is stored in AQvBytes).
Bytes	8-bit mask of bytes used with the addressed doubleword. (Approximate) A byte mask is used to determine if dependencies exist between load and store instructions which access the same double word in the cache. For simplicity, load or store "left/right" instructions are assumed to use the entire word. This may cause spurious dependencies in a few cases, but the only effect is to delay the load instruction.
DepC	16 by 16-bit dependency matrix identifies all previous entries which use the same cache set. Discussed below.
DepS	16 by 16-bit dependency matrix identifies all previous entries which cause a store-to-load dependency (i.e., between stores and subsequent loads). Discussed below.

Table 2: Address Queue Dependency Bits

As discussed below, Index [13:3] and bytes are also held in address stack 420. However, the byte mask contained in address stack 420 is more precise than that held in address queue 308.

Table 3 lists control bits, which are determined during the course of instruction execution and held in segment 517, shown in Fig. 7.

Field (AQv-__)	Description (Each field is 1 bit)
Calc	Address has been calculated in the address calculation unit 418. This bit is set at the end of cycle "E2" of the address calculation sequence. The address must be calculated before any other operation can be completed.
TagCk	Entry has completed a tag check cycle with data cache 424. If it resulted in a "cache hit", the operation may proceed. Otherwise, the entry will wait until it can be executed sequentially. (i.e., If a retry is necessary, it is delayed until all dependencies are removed.)
Wait	This entry set its "hit" state bit but that bit was later reset by an external interface command from unit 434. Unless an exception was also set, this entry must retry its tag check cycle to fetch a new copy of its block. However, its request must wait until this entry becomes the oldest entry within the address queue 308. This bit set if a "mark invalid" command matches any entry, or if a "mark shared" command matches an entry containing a store instruction. The entry's request is delayed so that the cache will not be continuously thrashed by speculative reads.
Ref	Memory block is being refilled by external interface 434 into a primary cache (i.e., instruction cache 102 or data cache 424).
Upg	Memory block is being upgraded for a store instruction, so that it can be written. (This block's cache state will become "dirty exclusive, inconsistent".)
Hit	Memory block is in cache, or is being refilled into cache (if AQvRefill). More specifically, the "hit" bit is set if the queue gets a cache hit or if it initiated a refill operation for the addressed block. "Hit" only means that the entry's address matches a valid address in the cache tag; it can be set before the refill operation is completed.
Way	Way of the primary data cache in which this block is located. (When the queue gets a "hit" or initiates a cache refill, it records which way of the cache was used.)
Unc	This bit is set for cache-ops and loads or stores to addresses within "uncached" regions of memory. These instructions must be executed sequentially.
Done	This entry has been completed. (This bit is never set for store instructions.)

Field (AQv-__)	Description (Each field is 1 bit)
ExcCal ExcSoft ExcBus	This entry has an exception and will be aborted. ExcCal: The exception was detected during address calculation or translation. Invalid addresses or mis-aligned addresses are detected from the calculated address. Translation exceptions include TLB miss, invalid translations, or write protection violations. ExcSoft: A soft exception is detected when external interface 434 invalidates a cache block used by a load instruction which is already "done" but has not yet graduated. (Soft exceptions flush the pipeline to clear any use of stale data, thus preserving strong memory consistently, but they are not visible to the program.) ExcBus: The exception was detected when external interface 434 signalled a bus error for a memory operation initiated by this entry.
BusyS	Stage "C1": Entry is busy. Either: 1) The entry was issued to the address cal-culation unit 418 during previous cycle, or 2) The entry is being retried by address queue 308.
BusyT	Stage "C2": Entry is busy.
MatchS	Entry is in pipeline state "C1" following end of refill state.
MatchT	Entry is in pipeline state "C2" following end of refill state.
MatchU	Entry is in pipeline state "C3" following end of refill state.
LoadReq	Entry requested a "freeload" cycle within the last three cycles. If this request was not granted, it will request a "justload" cycle instead.
LockA LockB	This entry has completed a tag check cycle, and has "locked" the block it needs into its cache set. Blocks can be locked out of a program sequence, but only one block can be locked per cache set.
UseA UseB	This entry is using a block which is not (usually) locked in the cache. This "use" is permitted only for the oldest entry within each cache set.

Table 3: Address Queue Control Bits

The "C" numbers referenced in Table 3 identify pipeline "cycles" or "stages" associated with cache operations. As discussed below, address calculation uses a three-stage pipeline (i.e., C0-C2, and writes the result into

address stack 420 during a stage C3). Similarly, "E" numbers identify pipeline "cycles" or "stages" associated with execution operations. Generally, cycles E1, E2 and E3 correlate with pipeline stages 4, 5 and 6, identified in Fig. 1. "E" and "C" numbers are essentially interchangeable.

Busy signals AQvBusyS and BusyT prevent any entry from requesting while it is still in the pipeline. Two entries can be busy simultaneously, whenever the queue calculates the address for one entry while it retries a second entry. These entries can be distinguished, because the AQvCalc bit has not yet been set within the first entry.

The "refill operation" referenced in Table 3 (in conjunction with match signals AQvMatchS, MatchT and MatchU) is the process of loading new data into a cache block. More specifically, if a load instruction gets a "miss" from data cache 424, it waits until the cache is refilled. During a primary cache refill, an 8-word data block is read from either secondary cache 432 or main memory and is written into data cache 424. Each block is transferred as quadwords (4 words or 16 bytes) on two cycles.

Refill state is reset after data is refilled into data cache 424 unless there was a secondary cache miss or ECC error. (A 9-bit "Error Check and Correction" (ECC) code is appended to each 128-bit doubleword in secondary cache 432. This code can be used to correct any single-bit error and to detect all double-bit errors.)

The three "block address match" signals are pipelined into stages "C1", "C2" and "C3." A miss or an error is detected during "C2". If neither occurs, refill state ends during phase 2 of cycle "C3."

The "freeload" cycle referenced in Table 3 (in conjunction with signal AQvLoadReq) is when address queue 308 bypasses data to register files 302 or 306 for a "load" instruction, while external interface 434 is writing this same data into data cache 424 during a cache refill. This "free" load avoids using an extra cache cycle for the load. (Only one register can be freeloaded per cycle.) However, if a requested freeload cycle is not granted, the entry will request a

"justload" cycle. This cycle reads only the data array 425 of data cache 424 -- tag array 426 is not used.

The AQvLoadReq bit overrides the "refill" state to enable the freeload and justload requests for three cycles. After this, the "refill" state will have been reset, unless there was a secondary cache miss or ECC error.

The last entries in Table 3 (i.e., AQvLockA, LockB, UseA, UseB) are used to prevent cache thrashing. Once an entry either "hits" or "refills" a cache block, that block is kept in the cache until the associated entry graduates, unless it is invalidated by external interface 434. This procedure depends on the 2-way set-associative organization of data cache 424. That is, there are two blocks per cache set. The first entry to access a cache set may "lock" its block into that set. It remains locked as long as it is addressed by any active entry in queue 308. To prevent thrashing, the other block can be used only sequentially by the oldest entry which addresses that cache set. This is determined using AQvDepC (Table 2). This oldest entry marks its block as "used." (The "use" bits are required to keep the block in the cache. The "lock" bits could be reset if the locking instructions are aborted by a reversed branch.)

1. Address Queue Function Codes

Fig. 10 defines the 7-bit operation codes stored in the address queue 308 at segment 512 (as shown in Fig. 7). This code is condensed from 17 bits of the instruction through decode logic 505. The top two bits of the condensed function code indicate the type of function. (The subfield which is stored in the queue is underlined in the "Function" column of Fig. 8.)

"Onnnnnn" Bits 5:0 are a major opcode, from bits 31:26 of the original instruction. The opcode may have been modified during predecode. These opcodes include most load/store instructions.

"10nnnnn" Bits 5:0 are a COP1X function code, from bits 5:0 of the original instruction. These instructions use base-index addressing for floating-point registers or for prefetch.

"11nnncc" Bits 4:0 are a CACHE operation code, from bits 20:16 of the original instruction. The three high bits specify an operation; the low two bits select a cache.

The top section of the table shows modified opcodes (Instruction bits 31:26). Several opcodes are modified during predecode to facilitate decoding. The original opcodes of these instructions are shown in parentheses, as "(op=32)"; their previous positions are marked by their mnemonic in small italic letters, as "*(ldr)*". All modifications are limited to the high three opcode bits, so all changes are within a column. "LWC2" is mapped into the same box as "SWC2"; both instructions cause the same exceptions and do not load any destination register.

The COP1X opcode (octal #23), which is split and moved to two separate opcodes (octal#13 and #33) during predecode, and the CACHE opcode (octal #57) are replaced with their function and subfunction fields, respectively. Thus, these codes are not stored in the queue. (The corresponding boxes are shaded to indicate that these codes do not occur.)

The middle section of the table contains five function codes from "COP1X". These do "Load Indexed", "Store Indexed", and "Prefetch Indexed" operations. The function code is instruction bits 5:0. The two "Load" instructions are moved to #33, because they have a floating-point destination register. The prefetch and the two "Store" instructions are moved to #13, because they have no destination register.

The bottom section contains cache operations. There are eight operations which operate on the Instruction Cache (IC), Data Cache (DC) or Secondary Cache (SC).

There are two formats of "prefetch" instructions: the "LPF" opcode (#63 changed to #73) and the "COP1X" function "PFETCH" (#17).

2. *Address Queue Operand Registers*

Each address queue entry contains up to three operands. Operand A, stored in segment 509 (Fig. 7), is the integer register which contains the base address. Operand B, stored in segment 510, is an integer register. For integer "load modify" instructions (such as LWL and LWR), it contains the old destination register number. For integer store instructions, it contains the value to be stored. For indexed floating-point load/store instructions, operand B contains the index register. If "Register 0" was selected for either operand A or B, the operand is not "Valid", and its value is zero.

Operand C, stored in segment 513, contains a floating-point register number. For a floating-point store instruction, it is the register value to be stored. For a floating-point "load word" instruction in Mips-2, it contains the old destination register number.

Operands A and B each have a "Ready" bit (see Fig. 8), which indicates if the operand value is available yet. These bits are initialized from a Busy Bit Table (not shown), during the cycle following decode. This bit can later be set when the operand is written, using an array of associative comparators.

Operand A must be ready before issuing integer load/store instructions for address calculation. For floating-point, both operand A and B must be ready. (Both operands are used for indexed load/store; operand B defaults to "ready" for other floating-point load/store.). Before a stack operation can be requested, operand B (for integer) or C (for floating-point) must be ready. These operands are needed for "load/modify" instructions.

The ready bits are not checked when graduating store instructions, because all previous instructions must have graduated before the store is eligible to graduate.

3. Address Associative Ports

The Address Queue has two associative ports which compare cache index addresses. Each port uses dynamic comparators which are clocked on the leading edge of phase 2 (i.e., $\phi 2$) of the processor clock.

The dependency port compares the calculated memory address (VAdr[13:5]) generated by address calculation unit 418 to each entry of segment 516 (Fig. 7) to detect accesses to the same cache set. These comparisons occur during stage C1 of each address calculation sequence.

An external port 525 determines if an external request affects any entries in the stack. It compares the cache index (VAdr[13:5]), the doubleword address (VAdr[4:3]), and the cache way. These comparisons occur during stage C0 of each external operation sequence.

B. Address Calculation Sequence

Address queue 308 issues instructions to address calculation unit 418 when its base (and index for indexed load and store instructions) is ready. This sequence is illustrated in the timing chart shown in Fig. 11.

Address calculation requires a three-stage pipeline; i.e., stages C0 (issue), C1 (address calculation) and C2 (data cache) as shown in Fig. 11. This calculation is performed only once for each load/store instruction.

During phase 1 of stage C0, address queue 308 issues the oldest entry with a pending request. During phase 2, the base (and index) registers of the instruction are read from integer register file 306. At the end of cycle C0, the base register is latched in operand "A" and either the index register or the 16-bit immediate field is latched in operand "B." Either register may be bypassed from any of the three write ports 526-528 of integer register file 306 (Fig. 7).

For speed, data may be "bypassed" around a memory when reading a location during the same cycle that it is written. In other words. the newly written

value is multiplexed directly to a read port without waiting for it to be actually written into the memory. A bypass multiplexer is selected whenever the operand register number of a current instruction equals the destination register number of a previous instruction. Bypassing is necessary to reduce instruction latency.

During phase 1 of stage C1, the address is calculated using 64-bit adder 529 disposed in address calculation unit 418 (Fig. 7). For "base+offset" format, adder 529 adds the base register on line 530 to the sign-extended 16-bit immediate field on line 531. For "base+index" format, adder 529 adds base register on line 530 and index register on line 532 together. The resulting virtual address on line 533 is sent to address queue 308 (i.e., segments 515 and 516) and TLB 422, as shown in Fig. 7.

During phase 2 of C1, TLB 422 compares "virtual page" address bits 63:62 and 43:13 to each of its entries. Also during phase 2, address queue 308 compares "index" bits VAdr[13:3] to the index stored in each entry in segment 516. This helps determine cache block and store-to-load dependencies, as discussed below. The newly calculated index is written into the new entry in segment 516.

During stage C2, a physical address is output from the entry of the TLB that matched.

Stage C3 (write result) is used for writing a calculated physical address to address stack 420 (i.e., segment 519 as shown in Fig. 7).

C. Issuing Instructions to the Data Cache

Data cache 424 is the subject of a copending application, as noted above, and therefore will not be discussed in detail. However, as background, the data cache contains 32K-bytes of memory. It is interleaved using two identical 16K-byte banks (i.e., banks #0 and #1). Each bank consists of a 256-row by 576-bit data array and 64-row by 35-bit tag array. The data array can access two 72-bit doublewords in parallel. The tag array can access two 32-bit tags in parallel.

The banks operate independently. For some operations, the tag and data arrays can operate independently. Thus, there are four arrays (two tag and two data) which are separately allocated.

The arrays are allocated among requests from four circuits. All four circuits can operate simultaneously, if they are allocated the cache array(s) they need. The highest priority request is used for external interface. The next-highest request is used for graduating store instructions if oldest in active list 212. The next-highest request is used for all other load/store operations which are retried from the Address Queue. The next priority is for instructions whose address is being calculated. The lowest priority is for a store that is not the oldest instruction in processor 100. (See Table 4).

Each bank is 2-way set associative. Two tags are read and compared in parallel for tag checks for the CPU and external "interrogate" operations. This determines which way of the cache, if any, contains the desired data. The way is remembered and used later for graduating stores, or for external refill or writeback operations.

The data cache is addressed by virtual address bits. Address bits [2:0] select a byte within a doubleword. Bits [4:3] select a doubleword within a block (8-words). Bit 5 selects bank #0 or bank #1. Bits [13:6] address a block within the bank. These 8 bits are decoded to select one of 256 "word lines" in the cache data array. Each word line enables eight doublewords (16 words). Thus, the word line enables two blocks which represent a single set in a 2-way set-associative cache. Doublewords within these blocks are selected using 4-to-1 multiplexers in the sense amplifiers. Bits are interlaced so that the cache can access doublewords differently for processor or external interface operations. The processor associatively accesses doublewords within two blocks. The external interface accesses two doublewords within the same block in parallel.

As noted above, data cache 424 is shared between three processor units and external interface unit 434. The processor units perform tag checks, load instructions, and store instructions. These units compete for cache and register

resources based on priorities described below. Each unit can do only one operation per cycle, but all four units may operate simultaneously, if their resource requirements do not collide.

All load and store instructions are put in the address queue 308 after they are decoded. When their base and index registers are available, they are issued to address calculation unit 418 (Fig. 7), which computes a virtual address (VAdr[63:62,43:0]), and the TLB 422 which converts this into a physical address (PAdr[39:0]). These addresses are written into address stack 420. The "index" bits of the virtual address (i.e., VAdr[13:0]) are written into address queue 308 (at segments 515 and 516), for dependency checking. This sequence is performed by a dedicated pipeline.

Data cache 424 can be accessed in parallel with a TLB access, if the required bank's tag array is available. If the tag array generates a cache "hit" for a load instruction, that instruction can be completed immediately. Otherwise, or if there are "store-to-load" dependencies, the instruction must later be re-issued from the address queue 308. These later accesses can do tag checks, loads, or both. A separate unit controls writing register values into the cache, as part of graduating store instructions.

1. Data Cache Usage

As noted above, data cache 424 contains two identical banks. Each bank contains tag and data arrays which can operate separately. Thus, there are four separate cache arrays which can be allocated independently. Each request contains a 4-bit code which indicates which of these arrays it needs:

...**UseC[3]:** Bank 1 Tag Array.

...**UseC[2]:** Bank 1 Data Array.

...**UseC[1]:** Bank 0 Tag Array.

...**UseC(O]:** Bank 0 Data Array.

Usage signals are generated for the external interface (ExtUseC), queue retry (AccUseC), and store (StoUseC) during the request cycle ("C0"). This cycle is two cycles before the array is read or written. If none of these requests occur, the arrays are available for use by the instruction whose address is being calculated.

Priority for using the Data Cache is illustrated in Fig. 12 and Table 4. Each unit can request an operation from either Data Cache bank, depending on its address bit 5. Some operations require use of both the tag and data arrays within that bank. High priority requests are determined during the "C0" request cycle. Lower requests are not determined until the next cycle, because they depend on signals during "C1".

External interface 434 provides a 4-bit command code ("DCmd[3:0]") and a cache index ("Index[13:4]") to the processor (i.e., cycle C0), two cycles before it uses data cache 424. The command and address bit 5 are decoded to determine which cache arrays are needed. The external interface has highest priority; its requests are always granted. If this command refills a data quadword, and address queue 308 contains a "load" instruction in an entry waiting for this quadword, the queue selects and issues that entry to the load unit. This operation, as described above, is called a "freeload."

Store operations are requested only if the oldest store instruction is ready to graduate, its store value is ready from a register and any previous load has been completed without an exception. (The request for a shared read port is made one cycle earlier. This request has low priority, because it is unknown if the store could graduate.) If this store instruction is the oldest instruction (not just the oldest store instruction), it is given high priority for the cache, because it is guaranteed to graduate if ready. Otherwise, other queue accesses are given priority.

Queue retry accesses can perform tag checks, loads, or both. Each entry can generate one of four requests, based on its state and bit 5 of its address. For each bank, these include a request for the tag array (and perhaps the data array)

and a request for only the data array. Address queue 308 uses four priority encoders to select one entry from each group of requests. One of these four is then selected, after it determines which arrays were needed by the external interface and guaranteed stores. Tag checks are given priority over data-only requests. Priority between the banks is alternated. (This makes use of the two banks more uniform. Also, if a tag check generates a refill, that bank's tag array will be busy during the next cycle, which would abort any new operation which uses that tag array.)

Lower-priority requests are resolved during "C1" cycle. Active list 212 determines which instructions graduate. If a store instruction graduates at the beginning of the "C1" cycle, it will use its data cache bank. For instructions whose address is being calculated, the bank is selected using VAdr[5], which becomes valid at about the middle of phase 1. A tag check cycle is performed, if the selected tag array is available. For load instructions, a data load operation is also performed, if the data array is also available. (It is useful to complete the tag check, even if the load must be repeated. If the tag check results in a "miss" or a memory dependency, the data array would not be used anyway. If it hits, the next retry from the queue will need only a data array.)

Pri-ority	Unit	Description
1	External Interface	External interface may preempt use of either bank of the data cache 424 by asserting a request on two cycles before the cache access.
2	Certain Store	A store instruction is given priority if it is guaranteed to graduate.
3	Retry Access	Instruction retrying access has priority over new address calculation.
4	Address Calculate	Until a new address is calculated, the priority circuit does not know which cache bank it requires. It will acc ss the cache if the required bank is available.

5	Other Store	If a store is not the oldest instruction in the processor, it might not graduate. Thus, it has low priority so that it will not interfere with operations which are known to be needed.

Table 4: Priority for using Data Cache

2. Shared Register Usage

Most processor requests require use of a shared register port. These ports are allocated independently of the cache, by a move unit (not shown). The move unit transfers data between register files 302, 306 and data cache 424 and branch unit 214. These units share control of one read and one write port to each of the integer and floating-point register files. These shared ports are used for all register accesses except for the dedicated ports assigned the integer and floating-point arithmetic units. An instruction cannot be completed if it does not get the register ports it requires. However, a tag check can be completed.

Most loads require a destination register in either the integer 306 or floating-point 302 register files. If an integer load specifies "register 0." however, no destination is stored. "Load/modify" instructions also require a read port to fetch the old value of the destination register. The required registers are determined in the decode unit, and are signaled to the queue as operand "valid" bits.

For uncached load instructions, registers are allocated at the end of the "C0" cycle. These requests have absolute priority, because these instructions cannot be retried. For other load instructions, the ports are allocated during "C1". These requests have low priority.

For a store instruction, the data register is located and the appropriate register file port is allocated one cycle before "C0". Stores have the lowest priority for ports, because it is not known if they can actually graduate. Stores are selected by default if no other request is present.

3. *"Active" Signals in Address Queue*

Each queue entry is "active" if it contains a valid instruction. For timing reasons, there are several sets of "active" signals. These signals are generated using read pointer 506 and write pointer 507 (Fig. 7) of the address queue. Only the primary "active" signals correspond exactly to the pointers. The other signals vary slightly logically, but have better timing for special uses. The associated logic 1200 is illustrated in Fig. 13. Active signals are defined as follows:

AQvActive[15:0]: Active bits determined directly from the "read" and "write" pointers. These signals become valid late during phase 1. ("Primary" signals.)

AQvActiveB[15:0]: AQvActive, but delete entries which are aborted if a branch is reversed. These signals become valid late during phase 2.

AQvActiveL[15:0]: AQvActiveB delayed in a phase-2 latch. These signals become valid at the end of phase 2, and can be read dynamically using a phase-1 strobe.

AQvActiveF[15:0]: AQvActiveL delayed in a phase-1 latch. These signals switch cleanly at the beginning of each cycle. They are used to reset rows and columns of the dependency matrices discussed below. (The two latches create an edge-triggered register.)

Referring to Fig. 13, new decoded entries to address queue 308, identified by a 4-bit "load/store instruction mask," are processed by logic block 1201. Block 1201 generates a 16-bit "PutVect" signal which maps one of four input instructions to each of the address queue write ports. Block 1201 also informs logic block 1203 of the number of instructions written into address queue 308 each cycle via signal "InstrWr." Block 1203 calculates the next value of write pointer ("WrPtr") using the InstrWr signal. This block also calculates the next value of read pointer ("RdPtr") using the "mask of graduating load/store

instructions" (i.e., a signal indicating the number of instructions graduating from address queue 308). The RdPtr and WrPtr signals are forwarded to logic block 1204 which generates read mask 1205 and write masks 1206 and 1207. (Mask 1207 is the same as 1206 except aborted entries may be deleted if a branch is reversed via MUX 1208.) These signals are further processed in block 1204 to generate "active" signals, discussed below. In the event of a reverse branch, branch unit 214 provides information to restore the WrPtr to its pre-branch state. Major elements of the conventional logic used to implement these blocks is illustrated in Fig. 13.

As discussed above, RdPtr and WrPtr are 5-bit counters which configure the queue's entries as a circular FIFO. The four lower bits select one of the queue's 16 entries. The fifth bit distinguishes between an empty queue (read and write pointers are identical) and a full queue (pointers differ in high bit only). This bit also indicates if the write pointer has "wrapped" to zero module 16, and the read pointer has not. In this case, the write pointer is less than or equal to the read pointer.

As shown in Fig. 13, "active" bits are generated from two masks formed from the two pointers. Each mask sets all bits lower than the pointer, as shown in Fig. 14. For example, when the read pointer equals '6', its mask has bits [5:0] set. The "active" bits are set for entries which have been decoded ("written" WrMask) but not graduated ("read" ~RdMask). If the write pointer has not wrapped, the masks are logically ANDed. If it has wrapped, the masks are logically ORed.

The write pointer is the counter output directly. It is incremented at the end of each cycle by the number of load/store instructions which were decoded in block 1203. These instructions are written into the queue, and become active, at the beginning of the next cycle.

If a speculative branch was mispredicted, the write pointer will be restored using a shadow register associated with that branch from branch unit 214. This deletes all the instructions which were speculatively decoded after the branch.

The shadow register is loaded from the write pointer (plus the number of load/store instructions decoded before the branch, if any), when the branch is originally decoded. There are four shadow registers (i.e., 1209-1212), because the decode unit can speculatively fetch past four nested speculative branches. The restore signal (i.e., "RstrEn") is not valid until early in phase 2, so the write pointer and active mask are not updated until the next cycle.

However, the restore does delete entries from a later "active" signal; i.e., AQvActiveB[15:0]. These signals are logically ORed with signal "NewEntry[15:0]," which indicates newly decoded entries. The results are saved in a register composed of latches 1213 and 1214. The output of latch 1213 (i.e., AQvActiveL[15:0]) becomes valid late during phase 2, and can be read dynamically during phase 1 of the next cycle. The output of latch 1214 is valid during the following cycle. These signals are used to reset rows and columns of the dependency matrices. These signals are logically equivalent to the original "active" signals, except that they go inactive one cycle later after an instruction graduates.

The read pointer is the output of adder 1215 in logic block 1203, which increments the read counter ("RdPtrD") by the number of load/store instructions which graduated at the end of the previous cycle ("GR1WOGradLS"). These signals occur too late to be added before the clock edge.

The pointers are subtracted to determine the number of empty slots in address queue 308 in logic block 1202. This information is sent to the decode logic 200 (Fig. 1), so it will not decode more load/store instructions than will fit in the queue. The subtraction uses 1s-complement arithmetic to simplify decoding. The signals "AQ0D0EmptySlot[3:0]" are a unary mask.

4. Priority Logic within Address Queue

The Address Queue's priority logic is illustrated in Fig. 15. In this illustration, logic is arranged according to its pipeline position. This logic corresponds to four pipeline stages. The first stage locates the oldest and next oldest store instructions. The second "Request" stage allocates the cache arrays for external interface, guaranteed store, and retry accesses. The third "Set-up" stage allocates arrays for other stores which graduate, and for instructions in the Address Calculation unit. Fig. 15 shows the timing and relationship between major signals.

A "...UseR" signal contains four bits which indicate which shared register file ports of register files 302 and 306 are needed.

Bit 3: Floating-point Register File, shared write port.

Bit 2: Floating-point Register File, shared read port.

Bit 1: Integer Register File, shared write port.

Bit 0: Integer Register File, shared read port.

The External Interface Command code ("EX0DCmd" in Fig. 15) is decoded to determine which data cache sections are required by external interface 434 ("ExtUseC"). This code also indicates if refill data is available.

Requests for retry accesses of the Address Queue are determined during cycle C0. There are three groups of requests which compete: "freeload" (i.e., a load instruction in the queue can use data which is being refilled into the data cache 424), "retry access" (i.e., an instructions already in the queue can request that its operation be retried, if there are no dependencies, the required cache section is available, and the addressed block is not waiting for a refill to be completed) and "address calculate" (i.e., the cache can be accessed in parallel with address calculation and translation).

Freeload operations are given priority because they share use of the cache bank which the external interface is refilling. These operations do not have to compete for cache resources. However, they do have to compete for register

ports. The operation may also fail if the secondary cache misses or has an ECC error.

Retry access uses four sets of requests which correspond to the data cache sections, as described above with respect to the "UseC" signal. One of these sets is selected, based on which sections are not used by external interface 434 or a store. Thus, each set of requests is enabled only if the corresponding cache section is available. This resource check only considers one section for each request. If both the data and tag arrays are needed, only the tag array is checked. The tag array is used for tag check operations. If the data array is also available, it can be used for loading data. But the tag check can be done even if the data array is not available, and the data array will be accessed independently later.

Newly calculated instructions have the lowest priority. First they tend to be newly decoded instructions, and older instructions are usually given higher priority. The older instructions are more likely to have later instructions dependent on them and are less likely to be speculative. Also, the bank which they require is not know during cycle "C0"; it depends on address bit VAdr[5], which is calculated early during cycle "C1". Thus, the availability of its cache sections is not known until that cycle.

These three sets of requests are combined into a single request at the end of cycle "C0" identified as "AccComReq," as shown in Fig. 15. The highest priority request in the selected set is granted by a dynamic priority encoder at the beginning of cycle "C1". This encoder gives higher priority to older instructions, based on the read pointer (AQvRdPtr) of the address queue. (Generally, performance is improved by giving older instructions priority. Also, this avoids a possible deadlock case. In some cases, an entry will continually retry its operation every three cycles while it waits for a resource. This occurs, for instance, for "load-modify" instructions if the old value of their destination register is still busy. In rare cases, three such entries could monopolize all queue issue cycles, inhibiting lower priority instructions. If priority was not based on

program order, this would create a deadlock if one of the inhibited instructions generates the old destination value.)

External interface 434 also sends the cache index address (i.e., "CC0PDIndex[13:4]" and "CC0PDWay"). If these match any queue entry which is in "refill" state, that entry's refill bit is reset. This allows that entry to request a load operation. For a load instruction, the refill data can be bypassed to the processor while it is being written into the cache. Because this bypass does not require an extra cache cycle, it is called a "freeload" operation, as described above. The refill may be aborted during the "C2" cycle if there was a Secondary Cache miss or an ECC error. If aborted, each corresponding entry must have its refill bit set again (AQvRefill). Thus, the match signals are pipelined in each entry (AQvFreeS and AQvFreeT; i.e., the freeload request signal is pipelined in analogous fashion to the match signals described above). If the refill is aborted during a freeload, the abort signal inhibits a "LoadDone" signal of the load instruction. If the load is retried later (but was requested before aborting), the AQvRefill bit of the subject entry is inhibited.

5. Access Priority Logic

Older instructions have priority for retry accesses in the address queue 308. The read pointer 506 of the queue (Fig. 7), which points to the oldest instruction in the queue, selects the entry with highest priority. Subsequent entries have decreasing priority. An example is illustrated in Fig. 16. As shown, entry 9 is the oldest. Entries 10 through 15, and then entries 0 through 8, are next oldest. Entry 8 contains the newest instruction and has the lowest priority.

This logic is implemented as shown in Fig. 17. The sixteen entries are divided into four groups of four entries. Each group is implemented with identical logic. The grant signal has two inputs. The first input (upper and-or input in drawing) is asserted by any request in the highest group of requests. This group is selected by decoding the upper two pointer bits: RdPtr[3:2]. This group

may also contain several of the lowest requests, which are gated off using a bit mask generated from RdPtr[1:O].

Lower priority requests are granted using a 4-wide and-or gate which determines if any higher-priority group contains any request. The grant for the "highest" group enables the lowest entries, which were masked off from the group requests. This priority circuit includes the "highest" bits, but it will not be enabled if any of those requests are pending.

The tables in Fig. 17 (i.e., 1610-1613) identify which bits within each group are enabled for a selected group. A number indicates the bit is enabled while an "x" indicates the bit is disabled. Accordingly, if Ptr[3:2] is "00" then, all table 1610 outputs (coupled to AND gates 1602-1605) are high. Tables 1611-1613 operate in similar fashion.

Logic 1600, shown in Fig. 17, implements "Group 0" (i.e., containing bits 3:0). This logic is replicated four times for sixteen bits (i.e., Groups 0 through 3). RdPtr[3:0] selects the oldest entry in the queue. Ptr[3:2] identifies the associated group and "High[n]" is active for the nth group (i.e., if Group 0 contains the oldest entry, then High[0] is a logic "1"). "Req[0]" to "Req[3]" are retry requests from entries 0 through 3, respectively, in a particular group.

Lines 1606 to 1609 (i.e., "Group0" to "Group3") are a logic 1 when a request is present in the respective group (i.e., each line is coupled to an OR gate which is, in turn, coupled to a circuit identical to logic 1600 for that particular group). When a request in a group is granted, the appropriate "Grant" signal (i.e., Grant[0], [1], [2] or [3]) is high. Additional circuit functional description is provided in Fig. 17.

6. Synchronize Instruction ("SYNC")

A "SYNC" instruction ("SYNC", opcode 0 with function octal '17') provides a memory barrier, which may be used to control sequencing of memory operations in a loosely ordered system. Architecturally, it guarantees that the

entire system has completed all previous memory instructions, before any subsequent memory instruction can graduate. Write-back buffers (for implementing a write-back protocol) of external interface 434 must be empty, and the external memory system of processor 100 has no pending operations. External interface 434 asserts signal "CD0SyncGradEn" whenever a SYNC instruction may graduate.

SYNC instructions are implemented in a "light-weight" fashion on processor 100. The processor continues to fetch and decode instructions. It is allowed to process load and store instructions speculatively and out-of-order following a "SYNC." This includes refilling the cache and loading values into registers. Because processor 100 graduates instructions in order, however, no data is stored and none of these instructions can graduate until after the SYNC graduates. One of these speculative loads could use a cache block which is invalidated before it is graduated, but it will be aborted with a "soft exception."

Whenever a primary data cache block is invalidated, its index is compared to all load instructions in address stack 420. If it matches and the load has been completed, a soft exception ("strong ordering violation") is flagged for that instruction. The exception prevents the instruction from graduating. When it is ready to graduate, the entire pipeline is flushed and the state of the processor is restored to before the load was decoded. Because this exception is soft (and must not be reported to the kernel), the pipeline immediately resumes executing instructions.

Address queue 308 continues to execute load and store instructions speculatively. If an external bus operation causes the needed cache line to be invalidated, the instruction will be aborted using the "soft" exception mechanism and then automatically retried.

A "SYNC" instruction is loaded into address queue 308, but the queue does not calculate an address or perform any operation. It is marked "done" after it becomes the oldest instruction in the address queue and external interface 434 asserts "CD0SyncGradEn." It can then graduate in order.

7. *Synchronizing Cache-op Instructions*

Cache-op instructions ("CACHE", opcode octal '57'; Fig. 10) are executed sequentially by the address queue 308. Whenever a cache-op instruction is in the queue, the execution of all later instructions is inhibited until it graduates. Address calculation is performed, but cache access is not enabled.

Whenever one or more cache-op instructions are in the address queue 308, they generate a mask which inhibits any subsequent instructions from completing a load instruction, or from requesting a retry access. The logic which generates this mask is illustrated in Fig. 18. This generates a mask which inhibits all entries after a sequential instruction. The 4-bit read pointer "Ptr[3:0]" selects the oldest instruction in the queue. The input "Sync[15:0]" is a 16 bit vector in which a 'l' indicates that the entry is a cache-op. This circuit sets a 'l' in all bits following the first input bit which is set.

The sixteen entries are divided into four groups of four entries. Each group is implemented with identical logic. The "SyncWait" signal has two inputs. The first input (upper and-or input in drawing) is asserted by any request in the highest group of requests. This group is selected by decoding the upper two pointer bits: RdPtr[3:2]. This group may also contain several of the lowest requests, which are gated off using a bit mask generated from Rdptr[1:0].

Like Fig. 17, the logic 1800 in Fig. 18 implements "Group 0," which contains bits 3 to 0. This logic is replicated four time for 16 bits. Only tables 1801, 1802, 1803 and 1804 change for each group. Ptr[3:0] selects the oldest entry. Sync[3:0] is input and SyncWait[3:0] is output. In tables 1801-1804, a number indicates an enabled (i.e., high) bit while an "x" indicates a disabled (i.e., low) bit.

Fig. 19 provides an example of the synchronize mask and Fig. 20 shows how the low pointer bits affect bits within the high group. Specifically, the "high" group contains four entries which may vary between the highest and lowest priority, depending on the low two bits of the pointer. When both pointer

bits are zero, all four entries are within the highest group and each can set mask bits for any entry to its left. For other pointers, some entries are within the lowest group. These entries can set mask bits within the lowest, but they cannot set any mask bits for the highest entries. All of the lowest mask bits are set if there is any bit set within any other group.

D. Retry Accesses

1. Access Request Logic

Each entry within the address queue 308 can generate a request to retry its operation. This logic is summarized in Table 5. Signals which are associated with retry accesses have a prefix "Acc...".

Request	16-bit Select	Description (In order of decreasing priority.)
UncLd-ReqEn *or* E0Unc-Load	RdPtrVec [15:0]	Begin an uncached load when it becomes the oldest instruction. Finish an uncached load when data is returned by External Interface.
UncStReq	StSel [15:0]	An uncached store instruction needs both the tag and data sections of the cache. It uses the tag section to send the address to the External Interface.
E0-GrantExt	ExtFreeReq [15:0]	A "free load" cycle is requested for a cacheable load instruction, while the cache is refilled from the External Interface.
Req-AnyAcc	AccReqMux [15:0]	An entry requests a retry operation for a cacheable instruction. The corresponding section of the data Cache is not being used by External Interface.
~Inhi-bitACalc	AQIssue [15:0]	An access is requested simultaneously with the initial address calculation, except when it is aborted or to re=calculate an address for an exception.
default	0	No requests.

Tabl 5: Cache Access Requests (AccC mReq)

Uncached Loads: Uncached load instructions are executed in program order. The request signal "UncLdReqEn" is generated when the oldest instruction in the processor is a load instruction whose address has an "uncached" attribute. This instruction is identified by active list 212. This instruction is requested in stage "C0" and is sent to external interface 434 during the tag check cycle in stage "C2". The instruction is limited to be in one stage at a time; new requests are inhibited while a previous request is still in stages "C1" or "C2". This request is always for the oldest entry in the queue, which is selected by queue's read pointer (RdPtrVec).

Uncached Stores: Uncached store instructions send their address to external interface 434 using the cache tag section. Thus, they require the access port of the queue as well as the store port. This request is for the oldest entry which contains a store instruction (StSel). This entry must be one of the oldest four entries within the queue. It is selected by the store logic.

Freeloads: Data may be bypassed directly to the processor, while it is being refilled into the Data Cache. Each refill addresses is compared to all entries during stage "C0". If this address matches any load entry which is in "refill" state, that entry requests a "freeload" access.

Access Requests: Each entry may request an "access retry" cycle, depending on its state and its dependencies on older instructions. It requests either a tag check or a data cycle. Either request is gated by the AccReqEn signal, which is summarized in Table 6.

Signals	Description
AQvActive	Entry is active. (It contains an active instruction.)
AQvCalc	Entry's address has been calculated. (Each entry is first issued to the Address Calculation unit to generate and translate its address. Subsequently, it can request a retry.)
~AQvBusyS & ~AQvBusyT	Entry is not in pipeline stage C1 or C2. (An entry can only have one operation in progress at a time.)

Signals	Description
~AQvDone & ~AQvExc	No further activity is allowed if an entry has already been completed, or if any exception was detected for it.
~AQvUnc	Cycles for uncached loads, uncached stores, or CacheOp instructions use separate request circuits.
~AQvRef & ~AQvUpg	No requests are made while the queue is waiting for the External Interface to complete a cache refill or upgrade operation.

Table 6: Enable Cache Access Requests (AccReqEn)

Requests are enabled only if the entry is active (determined from the queue pointers), and the address has been calculated. Each entry may have only one operation in the pipeline at a time. Thus, new requests are inhibited if this entry is busy in stages "C1" (S) or "C2" (T). An entry cannot request another cycle after its instruction has been completed ("done"), or it has encountered an exception. Requests are delayed if there is any cache set or store dependency. (Because addresses are calculated out of order, some dependencies appear only until previous addresses have been calculated.)

Requests are inhibited for uncached instructions or CacheOp instructions. These instructions use separate requests, which are described above. Requests are delayed while the entry is waiting for external interface 434 to complete a cache refill (AQvRef) or upgrade (AQvUpg) operation.

Signal AccReqEn enables a "first" tag check. This tag check cycle sets AQvTagck to indicate that this entry has already tried once. This bit may also be set during the initial address calculation sequence, if a tag check was completed. It is not set if the cache was busy, and the tag check could not be made. It is set if the tag check failed because of a memory dependency, however. This prevents the entry from being continuously retried when it has little chance of success.

If the latest tag check did not set the entry's "hit" bit it can request a tag check cycle again, if it has no cache or store dependency on a previous instruction.

When an entry requests a tag check, it also may use the data section of the cache if it needs it, and if it is available.

If a load instruction sets its "hit" bit, but it has not yet completed, it may request a data-only cycle, if it has no cache or store dependency on a previous instruction.

Address Calculate: If there are no other requests, access is granted to the entry (if any) whose address is being calculated. This provides data access simultaneously with the address translation ("C2" cycle). (Address Calculation logic 418 can use the tag section, if it is not used by a retry access. It does not require the access port to do this, because all the necessary signals come directly from address calculation 418.)

Detailed access request logic is shown in Fig. 21.

2. Retry Load Hazard

Certain external events could create timing hazards when the Address Queue retries a load instruction which already has a cache "hit". For example, an invalidate operation resets the "hit" bit in the queue for any entry with a matching address. If that entry is already "done", a soft exception must be set.

3. Dependency CAM Logic

Address queue 308 contains the index address field associated with each address entry. This logic is shown in Fig. 22.

The "block match" signal (DepMatchBlk) compares address bits 13:5. This signal is pipelined into stages "C1", "C2", and "C3". If the entry becomes not active, these later signals are inhibited.

The ExtMatch signal generates a request for a free load. It is generated if the entry is ready to do a load (AccReqEt), and its block, doubleword, and way bits match the corresponding bit from external interface 434. This signal sets a LoadReq flipflop, which is normally kept on for three cycles. This allows earlier access to the data.

E. State Changes Within Address Queue

1. Gating of Data Cache Hit Signals

Address queue 308 controls when Data Cache Tag logic does a tag check for any entry in the queue. Cache hit or refill can be inhibited by the queue's dependency logic.

If either way is "locked" into cache, that way cannot be selected for replacement.

If either way is "locked" into cache, and the entry has any cache dependency, the other way cannot be used. It is reserved for the oldest instruction which references that cache set. Thus. neither way of the cache may be refilled. The other way is inhibited from generating a cache hit.

Any refill is also inhibited if the external interface intends to use the same cache bank on the next cycle. Whenever a refill is initiated, the new tag must be written into the tag RAM during the next cycle. If this cycle is not available, no refill can occur.

F. Address Queue Timing

Address Queue timing is illustrated in Fig. 23.

All cache operations use three pipeline stages. Stage "C0" (or"R") requests access. Stage "C1" (or"S") sets up the address and write data. Stage "C2" (or "T") does reads or writes the tag and/or data arrays during phase 1. Tag

checks and data alignment occur during phase 2. For load instructions, a fourth stage ("U") is used to write the result into the integer or floating-point register file.

The "store" sequence requires an initial extra cycle ("CN") to find the oldest store instruction.

1. Address Queue Busy Mask

As mentioned above, address queue operations are pipelined in four 1-cycle steps:

"R" Cycle "C0" Request operation.

"S" Cycle "C1" Set-up cache.

"T" Cycle "C2" Tag check (and/or other cache operations).

"U" Cycle "C3" Update registers.

Each operation is requested during cycle "R" based on the entry's state. Its state is modified at the end of cycle "T", based on the result of the operation. Each entry is limited to a single, non-overlapped operation, so new requests are inhibited during that entry's cycles "S" and "T". These inhibits are implemented using two pipelined busy masks: AQvBusyS[15:0] and AQvBusyT[15:0]. AQvBusyS is set for an entry either when it is issued to address calculation unit 418, or when it is issued for a retry access. Thus, two bits in AQvBusyS may be set simultaneously. These will be distinguished by the state of the AQvCalc bit. AQvCalc is set after the address calculation finishes cycle "T" (regardless of whether it completed a tag check). Thus, AQvCalc is zero for the entry being calculated; it is one for any retry. AQvBusyT is simply AQvBusyS delayed one cycle.

The cache select signals are decoded during "E1" to determine which request generated either a tag check or data load cycle.

AccDoesTCN Instruction from address queue 308 will do tag check during next cycle.

AccDoesLdN	Instruction from address queue 308 will do data load during next cycle.
ACalcDoesTCN	Instruction from address calculation unit 418 will do tag check during next cycle.
ACalcDoesLdN	Instruction from address calculation unit 418 will do data load during next cycle.

These signals are delayed one cycle, for use during the data cache access. (For this cycle, omit the postfix "N" from the signal mnemonics.)

The entry being tag checked is identified using AQvBusyT during a tag check cycle. If the tag check was issued as a retry, there is only a single operation, and the mask has only one bit set. Otherwise, the tag check used the entry with AQvCalc zero.

III. Address Stack

Address stack 420 (Fig. 7) is logically part of address queue 308, but is physically separate due to layout considerations. The address stack contains the physical memory address for each instruction in address queue 308. This address consists of two fields, the physical page number (RAdr[39:12]) and the virtual index (VAdr[13:0]). These fields overlap by two bits because data cache 424 is virtually indexed (i.e., virtual bits [13:3] select a data doubleword in the data cache) but physically tagged (i.e., cache tags store physical address bits RAdr[39:12]).

The translated real address (i.e., RAdr[39:12]) is latched from TLB 422 (Fig. 7). The low 12 bits of the real address equal corresponding bits of the virtual address Vadr[11:0].

The low 14 bits of the virtual address (i.e., VAdr[13:0]) are latched from the calculated address. These bits select a byte within the data cache array. The low 12 bits are an offset within the smallest virtual page, and are not modified by TLB 422.

Address stack includes additional information such as "access byte mask" (indicating which of the eight bytes of the accessed doubleword are read or written), "access type" (indicating which type of instruction is being executed; i.e., load, store, etc.) and "reference type" (indicating the length of the operand.

The address stack is loaded during the address calculation sequence (Fig. 11). Thus, it has a single write port. It has two read ports. A "stack" port is used when address queue 308 retries an operation. A "store" port is used when a store instruction is graduated.

IV. Memory Dependency

This logic is implemented in address queue 308; associated with segment 515 shown in Fig. 7.

A. Memory Dependency Checks

Load and store instructions are decoded and graduated in program order. However, to improve performance, memory operations to cacheable addresses may be performed out of order, unless there is a memory dependency. There are two types of dependency. First, a true memory dependency exists between a load instruction and any previous store which altered any byte used by the load. Second, accesses to the same cache set may be delayed by previous accesses to other addresses which share that set. This is an implementation dependency which prevents unnecessary cache thrashing. It is also necessary for proper operation of the dependency check procedure.

In a cache, a "set" is the group of blocks selected by each index value. In a direct-mapped cache, this index selects a set consisting of a single block. In an "n-way" set-associative cache, this index selects a set of "n" blocks. Cache addressing is described above in Section II.C.

Although memory loads are performed out of order, processor 100 appears (to a programmer) to have strong memory ordering. It detects whenever strong ordering might be violated, and backs up and re-executes the affected load instruction.

Accesses to non-cacheable addresses are performed in program order, when the corresponding instruction is about to graduate. All previous instructions have been completed, so no dependency check is needed. This is discussed below.

Memory dependencies are resolved within the address queue 308. A dependency may exist whenever two real addresses access the same cache set. Dependency checks must use real rather than virtual addresses, because two virtual addresses can be mapped to the same real address. For timing and cost reasons, however, the 40-bit real addresses are not directly compared. Instead, dependencies are detected in two steps.

Address queue 308 contains two 16-row by 16-column dependency matrixes. These matrixes are analogous, except for the logic equations defining how bits are set. "Cache Dependency Matrix" 2400, shown in Fig. 24, keeps track of all previous entries which use the same cache set. "Store Dependency Matrix" 2450, also shown in Fig. 24, keeps track of all dependencies of load instructions on previous store instructions. Because store dependencies exist only between operations on the same doubleword (i.e., all memory accesses are within doublewords), bits set in store matrix 2450 are always a subset of those set in cache matrix 2400. The operation of each matrix is illustrated in Fig. 25.

In the first step of a dependency check, a 9-bit cache index (VAdr[13:5]) is associatively compared to each entry in address queue 308 (i.e., segment 516 of Fig. 7). This identifies all previous entries to the same cache set. This comparison occurs while the virtual address (VAdr[13:0]) is written into stack 420 at the end of the address calculate cycle. Each matching entry is flagged by setting a corresponding dependency bit.

Second, the translated real address (RAdr[39:12]) is associatively compared to the data cache address tags. If there is a hit on the same side (i.e., way) of the cache, the new address selects the same cache block. If there is a miss, the cache block must be refilled before all dependencies can be resolved. This tag check cycle usually occurs one cycle after the address calculation, but it may be delayed if the data cache is busy.

1. Dependency Checking if Virtual Coherency

The dependency circuit must function properly even if the program uses virtual aliases. A virtual alias occurs if two different virtual addresses are translated to the same real address. Aliases must be considered, because associative comparator uses two virtual address bits (VAdr[13:12]) as part of the translated real address. (The lower bits (11:5) are part of the page offset, which is the same in the virtual and real addresses.) If aliases differ in these bits, the dependency logic will mistake them for distinct real addresses, and will fail to detect any dependencies between them. However, Secondary Cache 432 stores the two "primary index" bits (PIdx; i.e., VAdr[13:12]) and generates a "Virtual Coherency" exception if a different index is used. That instruction will be aborted with a soft exception, so any dependency does not matter.

2. Cache Block Dependencies

Memory dependencies are resolved by comparing cache indexes and using the cache hit signals. This method requires that each address be brought into the data cache before all dependencies can be resolved. Once an instruction has used a cache block, that block must remain in the cache until that instruction has graduated. Although the processor does not invalidate any block that is still in use, external interface 434 may. If it invalidates a block which has been used to load a register before the load instruction has graduated, that instruction is

flagged with a soft exception (described above) which will prevent it from being completed.

Data cache 424 is 2-way set associative. That is, it can store two independent blocks in each cache set. One of these blocks may be used for out-of-order operations. The second block must be reserved for in-order operations within that cache set. This guarantees that the processor can complete instructions in order without having to invalidate any block while it is still in use.

If a third block is needed, there is no room to bring that block into the cache. So that instruction and all subsequent accesses to this set must be delayed until all previous accesses to this set have graduated.

3. Load Dependency on Previous Stores

Whenever data is stored and then loaded from the same location, the load must get the newly stored data. A memory dependency exists between a load and a previous store if:

a. Both reference the same cache block (i.e., the dependency bits indicate the same cache set; the load must have a cache hit on the same way as the store);

b. Both reference the same doubleword (i.e., address bits 4:3 are equal) and;

c. The byte masks have at least one byte in common.

Memory dependencies are detected during tag check cycles, because the cache hit signals are required to determine the selected way. The cache dependency mask of the load entry (i.e., mask 2501 in Fig. 25) is used to select entries. Each entry detects a store dependency if it has the same block, doubleword, and any of the same bytes. Otherwise, the corresponding dependency bit is reset. A load instruction may be dependent on several stores; it must wait until all have graduated.

Address queue 308 uses somewhat simplified decoding for determining byte masks. It classifies each instruction by length - byte, halfword, fullword, or doubleword. The utilized bytes are identified in an 8-bit byte mask (doubleword contains 8 bytes) generated by address calculation unit 418. For simplicity, "Load Word/Doubleword Left/Right" instructions are treated as if they used the entire word or doubleword, even though some bytes may not be needed. These instructions are infrequently used in normal code, so this has negligible impact on performance.

These checks cannot be completed if any previous store has not had its address calculated yet. In such case, the load will remain dependent on the previous store until the address of the store is calculated. If the new address selects the same cache set, the load will wait until the store instruction has graduated. Otherwise, the associated dependency bit is reset and the load can proceed as soon as all remaining dependency bits are reset.

4. Dependency Checks When Loading Addresses

Virtual address bits VAdr[13:0] are loaded into an entry in both address queue 308 and the address stack 420 (Fig. 7) during the second half of each address calculate cycle. (These bits are duplicated due to layout considerations for processor 100.) At the same time, the "index" address bits VAdr[13:5] are associatively compared to every other entry. These comparators define the initial value for the dependency bits. This dependency check is based solely on the cache index, because the cache hit signals are not yet known. Usually, the next cycle does a tag check which determines the cache hit signals. However, it may be delayed due to tag update cycles or external intervention.

When loading an address for entry "n" into address stack 420, its fifteen dependency bits (i.e., Dep[n][k=O..15, k≠n]) are set using the following method. As note above, the cache "index" is bits VAdr[13:5]. Each dependency bit Dep[n][k] is set high (i.e., to a logic "1") when:

(1) Entry "k" contains an instruction previous to entry "n" (as defined by the cache dependency mask); and

(2) The address for entry "k" has not been computed or index[n]=index[k].

The dependency bit of a previously calculated entry "k" may be reset. This dependency bit was previously set because the new entry's address had not previously been calculated. If entry "k" has index[k]≠index[n], then reset Dep[k][n]. (Note reversal of subscripts on Dep.)

Fig. 25 shows an example of newly calculated index 2502 (i.e., VAdr[13:5]) being loaded into entry #4. As represented by line 2503, this value is compared with every index entry (i.e., entries #0-#7) in address queue 308 via comparators 2504-2511. However, only those entries identified through mask 2501 to be "previous" instructions to entry #4 may be used to alter the dependency bits of entry #4. Accordingly, resulting comparison "states" 2512, 2513 and 2514 associated with previous entries #1-#3 may be used to set dependency bits 2516, 2518 and/or 2520 in matrix 2500 if entry #4 is dependent on entries #1, #2 and/or #3, respectively. Alternatively, if any previous entry (i.e., #1-3) has not yet been calculated, the dependency bit in entry #4 is set as a default, which can be reset when this earlier address is ultimately calculated.

Entries #5 and #6 contain later instructions that may depend upon entry #4. If, for example, entry #5 is loaded before entry #4, bit 2522 of entry 5 will be set. Although entry #4 is not yet calculated, dependency matrix 2500 follows a procedure that presumes dependency of earlier, uncalculated instructions until such presumption is proven false. Accordingly, bit 2522 may be reset when entry #4 is actually loaded. Of course, if entry #5 is loaded after entry #4, the standard operation described above controls.

For clarity, matrix 2500 shows only eight of the sixteen rows and columns present in the dependency matrixes of the preferred embodiment. Moreover, athough only a single matrix is illustrated in Fig. 25, the preferred embodiment uses two matrixes, as shown in Fig. 24.

5. *Dependency Checks During Tag Check Cycles*

Dependency is determined during tag check cycles using cache hit signals, and the state and dependency bits within the address stack 420, as shown in Fig. 26. This figure lists all legal combinations of input bits, which are not completely independent of each other. By means of explanation, abbreviations used in the figure are defined below:

"-": "don't care" for inputs, "no action" for outputs;

"D": cache block dependency (i.e., a previous entry uses the same cache set);

"L": indicates a store-to-load dependency, or possible dependency if previous store address not yet calculated;

"S": bit is set;

"C": bit is set conditionally, only if there is no store-to-load dependency ("L"=0);

Cache State: valid (V), refilling (R), not valid (N), valid or refilling (E); available (not refilling) (A).

Each entry includes five bits which indicate cache hit information. The two "Lock" bits (LockA and LockB) indicate that the subject entry is using the first (random order) block within the cache set, for either side A or B (i.e., way 0 and 1, respectively). This block is locked into the cache until all entries which use it have graduated. If new entries get a hit on a locked block, they will use it and set their corresponding lock bit.

The two "Use" bits (UseA and UseB) indicate that this entry is using the second (sequential order) block within the cache set, for either side A or B. This block is locked into the cache until this entry has graduated. Then this block may be replaced, if necessary, by the next sequential entry to access this cache set. If new entries get a hit on a "use" block, they cannot use it.

The "Dependency" signal indicates that this entry is dependent on a previous entry.

For each cache cycle scheduled by the processor, the corresponding entry in address stack 420 is read. This provides the address and dependency information.

The procedure must identify all entries which use the same cache set so it can determine if any block has already been locked. This identification uses the dependency bits 2402 in matrix 2400 (Fig. 24) to select all other entries with the same index, or whose addresses have not yet been calculated. Bits in both the row and column are used. For entry #n, row bits Dep[n][j] identify which other entries (#j) this entry is dependent on. Column bits Dep[j][n] identify other entries which are dependent on this entry. These bits include uncalculated entries -- which are reset when the address is calculated if it selects another cache set.

Before any cache block has been locked, every entry is dependent on, or depended on by, each other entry using the same cache set. When a block is locked, its dependency bits are no longer needed for sequencing. Instead, they simply identify all other entries using the same cache set. Thus, whenever any other entry selects the same set, it knows that the block is locked.

The dependency bits identify every entry which uses the same cache set, by logically ORing row and column bits. The row (bits within each entry) identify other entries on which this entry is dependent. The column identifies entries which are dependent on this entry. More specifically, the row and column dependency bits form 16-bit numbers which are ORed in a bit-wise fashion. Accordingly, the result is a single 16-bit number (i.e., mask) with a "1" in each position corresponding to an entry using the same cache set. This number is used to read lock/use array 2404, illustrated in Fig. 24.

Specifically, each bit that is set (i.e., a logic 1) in the foregoing 16-bit number (i.e., mask) enables the reading of LockA, LockB, UseA and UseB bits associated with that entry. Thereafter, all LockA bits are ORed together (i.e., column 2460) generating a single LockA value for the associated cache set. Similarly, all LockB bits (column 2462), UseA bits (column 2464) and UseB bits

(column 2466) are separately ORed to generate a single value for each status bit of the associated cache set. These bits indicate current activity on this cache set.

B. Dependency Logic

Fig. 24 shows dependency matrix 2400 and 2450 disposed in address queue 308. A set of comparators 2406 identifies dependencies for recording in each matrix, and an array of lock and use bits 2404 describe activity for each active cache set. DepCache[j] signals forwarded to matrix 2402 indicate cache set dependencies while DepBytes[j] signals forwarded to matrix 2450 indicate store-to-load dependencies. Each comparator in set 2406 is coupled to matrixes 2400 and 2450, like comparator 2408 (i.e., there are sixteen DepCache[j] and DepBytes[j] lines between comparator set 2406 and matrixes 2400 and 2450). Only a single connection is shown for purposes of discussion.

DepCache[j] signal on line 2410 indicates whether any dependency exists between an index address (i.e., cache set) stored in comparator 2408 and an address being newly calculated. Similarly, DepBytes[j] signal on line 2412 indicates whether any dependency exists between an entry stored in comparator 2408 and an address being newly calculated based on byte overlap. In Fig. 24, a newly calculated address is at entry 10; identified by the ACalcVec[j] signal gated by phase 2 of the processor clock (i.e., $\phi 2$) through NAND gate 2414 and passing through inverter 2416. Signal ACaclVec[j] identifies a newly calculated entry and enables initial dependencies to be written into the corresponding row. As discussed below, this signal also provides a means for resetting dependency bits in other rows which erroneously indicate a dependence on this entry (i.e., through column 10).

Referring to Fig. 24, the DepCache[j] signal on line 2410 indicates whether a cache-set dependency exists and passes this signal through every cell in row 7. This same signal is passed through every cell in column 7, as indicated by line 2418. Concurrently, ACalcVec[j] gated by $\phi 2$ provide a pulse (when both

signals are high) to all cells in row 10 (as shown by line 2420) and column 10 (as shown by line 2422). If entry 10 is dependent on previous entry 7, dependency bit at row 10, column 7 is set through the combination of signals on lines 2418 (DepCache[j]) and 2420 (ACalcVec[j]).

The DepCache[j] signal on line 2410 is also forwarded to MUX 2426 as a bypass to lock/use array 2404. A second input to MUX 2426 is provided by read bit line 2428, which contains both a row and a column. (Each read word line and read bit line in matrixes 2400 and 2450 contain both a row and a column.) Read line 2428 passes through latch 2430, which is gated by phase 1 of the processor clock (i.e., $\phi 1$). The MUX outputs a signal to lock/use array 2404 (through latch 2432 gated by $\phi 2$). This output value enables the reading of lock or use bits 2440-2443.

As noted above, dependency checking requires a two-step operation; i.e., comparing virtual address information in dependency matrixes (i.e., matrix 2400 and 2450) and comparing an associated translated real address with data cache address tags. In the course of the latter operation, the status of an accessed cache set is determined by reading any lock or use bits (held in array 2404) set by other entries accessing the same cache set.

Referring to Fig. 24, if the same entry is undergoing both steps of the dependency checking operation at the same time (i.e., virtual and real address comparing), signal ACalcDoesTC selects DepCache[j] through MUX 2426. If the entry associated with DepCache[j] matches the newly calculated entry (i.e., entry 10 in this example), DepCache[j] is high thereby reading out any associated lock or use bit (i.e., bits 2440-2443) in array 2404. An identical circuit consisting of a MUX and two latches is coupled to every row in Matrix 2400 enabling the same operation to be carried out in parallel. The net result is a 16-bit word (i.e., mask) defined by the contents of array 2400 that identifies any lock or use bits set (i.e., a logic 1) for the associated cache set.

Alternatively, if the same entry is not undergoing both steps of the dependency checking operation at the same time (i.e., there may be a pending

address for tag check operations at the time matrixes 2400 and 2450 are accessed), signal ACalcDoesTC selects read bit line 2428 (passing through latch 2430) with MUX 2426. Line 2428 enables the reading of certain dependency bits associated with the entry undergoing tag check operations. These bits are used to access lock and use bits in array 2404.

More specifically, the entry undergoing tag check operations enables signal TagEntryRd[j], which is gated by $\phi 1$ through NAND 2434 and passes through inverter 2436 as shown in Fig. 24. (In the example of Fig. 24, the entry undergoing tag check operations is entry 3.) TagEntryRd[j] enables the dependency bits located on the jth row and jth column of matrix 2400 to be read out. (As the "j" designation indicates, a TagEntryRd[j] signal is available for each row and column in matrix 2400.)

Referring to Fig. 24, read bit line 2428 combines with TagEntryRd to select dependency bits stored at bit locations 2444 (row 3, col. 7) and 2445 (row 7, col. 3). These values are ORed together and output to array 2404 through MUX 2426, enabling the reading of any lock or use bits that may be set if location 2444 or 2445 holds a set bit (i.e., logic 1). An identical operation is carried out for entries 0 through 2 and 4 through 15.

The net effect of this operation is to produce a 16-bit word consisting of all dependency bits on row and column j (i.e., row and column 3 in this example) ORed together in a bit-wise fashion. This word is then used to read out values from array 2404 that correspond to the associated cache set. (A TagEntryWrite signal (not shown) is used to set corresponding lock and use bits in array 2404 one clock cycle after the associated TagEntryRd signal.)

Any entries calculated prior to the newly calculated entry (i.e., entry 10 in this example) that could possibly depend on this entry will have previously set their dependency bit associated with this entry (i.e., bit 10) to a logic 1. As noted above, this is a default value when the earlier address is unknown. However, once the previously uncalculated entry is calculated, defaulted bit values may be reset if no dependency in fact exists. Referring to Fig. 24, line 2422 enables bit

values located in column 10 to be reset if the associated row had previously been calculated. In other words, the DepCache[j] signals associated with each row j indicate whether or not a dependency exists. Each such signal is combined with the signal on line 2422 (i.e., ACalcVec) to reset the corresponding bit 10 if no dependency exists.

In addition to the foregoing, Fig. 24 discloses signals DepRowC[j] and DepRowS[j] which are output from OR gates 2438 and 2452, respectively. These signals represent the ORed value of all dependency bits in an associated row. An analogous circuit is coupled to each row in matrix 2400 and 2450.

As noted above, matrixes 2400 and 2450 are identical except for the logic equations defining how bits are set. Further, matrix 2450 is not coupled to lock/use matrix 2404 nor are TagEntryRd[j] signals applied to its entries. Aside from these differences, the foregoing discussion related to matrix 2400 applies equally to matrix 2450.

1. Comparators

A logic block diagram of each comparator of Fig. 24 is provided in Fig. 27. (N.B.: Fig. 22 also shows a logic block diagram of address queue comparators combined with additional control signals.) Comparator 2408, for example, holds set index bits 2702, doubleword bits 2704 and byte mask value 2706 for the corresponding entry (the "index" address). These values were loaded at the time this entry was newly calculated. Through the use of comparators 2706-2716, these loaded values are compared with a newly calculated virtual address 2718, an associated byte mask 2720 and addresses from the external interface 2722, as shown in Fig. 27.

The newly calculated address 2718 and external interface address 2722 each compare set address bits 13:5 with index address 2702 using comparators 2712 and 2714, respectively. The calculated address 2718 also compares the doubleword within a cache block (bits 4:3) using comparator 2710 and checks for

byte overlap (derived from the operation type and bits 2:0) using comparator 2708. Bit 4 of the external address 2724 is compared using comparator 2716 to match quadwords during cache refill. The ExtCompQW signal is used to enable or disable the quadword comparison in connection with signal ExtMatch[j].

The comparison results are logically combined, as shown in Fig. 27, producing four signals. Signal DepBytes[j] is high when comparators 2708, 2710 and 2712 all identify matches; thereby indicating dependency on a particular byte. If the associated instruction is a load (identified by conventional logic not shown in this figure), then DepBytes[j] represents a "load dependency on previous store" and the signal is forwarded to matrix 2450. Signal DepCache[j] is high when comparator 2712 identifies a match; thereby indicating a cache set dependency which is forwarded to matrix 2400. Alternatively, if comparator 2408 is inactive (i.e., contains no values 2702, 2704 and 2706), CalcDep bit 2726 is reset thereby forcing DepBytes[j] and DepCache[j] high.

Signal ExtFreeReq is high when comparators 2714 and 2716 identify matches; thereby indicating a quadword match and enabling a freeload as requested by external interface 434. Finally, ExtMatch[j] is also high when comparators 2714 and 2716 identify a match (assuming ExtCompQW is low, otherwise only comparator 2714 effects output) thereby providing control information to queue 308.

Fig. 28 shows the circuit implementation of the dependency comparators of Figs. 24 and 27. Each entry has three comparators used with newly calculated addresses. The index comparator compares address bits 13:5 to determine if the new address selects the same cache set. The doubleword comparator compares address bits 4:3. The byte overlap circuit determines if the new byte mask selects any of the same bytes.

The index comparator is constructed with nine conventional comparator circuits 2800 shown in Fig. 28. Signal ACalcVec[j] identifies the comparator circuit associated with the newly calculated entry. This entry is stored in the circuit when first calculated and functions as the index address for subsequent

"new" addresses. These subsequent addresses are presented on bit lines 2801 and 2802, forcing line 2803 high if the new bit differs from the stored bit. If index address bits differ, a positive pulse coinciding with $\phi 2$ is presented on line 2804.

Similarly, the doubleword comparator is constructed with two conventional comparator circuits 2800 shown in Fig. 28.

Referring to Fig. 29, byte overlap circuit 2900 is constructed from stacks 2901 and 2902 of four transistors with complemented signals applied to their gates. Accordingly, a transistor is turned off only when an associated byte is present. If a pair of transistors are turned off (for example, transistors 2903 and 2904), line 2905 (i.e., ~DepOverlapB[j]) will remain low during $\phi 2$ thereby indicating an overlap (i.e., dependency).

These circuits switch dynamically on the phase 2 clock edge. Their outputs are pulses which switch about 3 inverter delays later. So that these pulses can be gated together without generating glitches, each circuit generates a pulse if there is no dependency. Comparator 2800 generates a pulse if any address bit differs, using standard comparator circuits and a dynamic "OR" (i.e., all related comparators coupled to match line 2806 via transistor 2808). The byte overlap circuit 2900 generates a pulse if no bit is present in both masks. This requires an "AND" circuit. It is impractical to wire a stack of 8 bits in series. so the outputs of two parallel 4-high stacks are ANDed. (A parallel "OR" gate would be faster and simpler, but it would generate a pulse if the two masks overlap.)

2. Dependency and Diagonal Cells

Fig. 30 shows the logic within the two types of cells included in matrixes 2400 and 2450. These cell types are diagonal cell 3002 (one per row) and dependency cell 3004 (15 per row). Each dependency cell 3004 stores one bit using cross-coupled inverters 3006. This bit is written from the comparator outputs (DepPrevC[15:0]) when the address is calculated. Bits can be reset when other addresses are calculated later, if it determines that there is no dependency

(DepCache[15:0]). Diagonal cells 3002 do not contain RAM bits, but they do connect horizontal and vertical control lines.

As shown in Fig. 30, signal TagEntryRd[j] enables the output of dependency cells in the associated row and column. This signal outputs a row dependency bit value (i.e., DepSet[k]) through transistor 3008, and a column dependency bit value (i.e., DepSet[j]) through transistor 3010.

Assuming dependency cell 3004 is in a row corresponding to a newly calculated entry, a bit (held by cross-coupled inverters 3006) may be written by enabling signals ACalcVec[j] (on line 3011), ~ACalcVec[j] (on line 3012), and data signal DepPrev[k] (on line 3014). As discussed above, ACalcVec[j] identifies the newly calculated entry. DepPrev[k] is a product of DepPrev[j] on line 3016 (which indicates whether an entry is previous to the entry being calculated; i.e., the mask of Fig. 25) and ~DepCache[j] on line 3018 (which indicates whether there is a cache set dependency with another entry (i.e., row)). These two signals are combined through NAND 3020 and inverter 3020, generating a signal that is input to transistor 3024, which feeds the signal to inverters 3006.

Alternatively, assuming dependency cell 3004 is in a row corresponding to a previously calculated entry, and this cell was previously set based on an earlier entry that had not yet been calculated (as discussed above), this bit may be reset (if there is no dependency) using ~DepCache[j] on line 3026 and ACalcVec[k] (on line 3028) generated from the now-calculated earlier entry.

Also shown in Fig. 30 is signal ~Active[j] on line 3030 which resets an entire row if entry [j] is not active. Similarly, signal ~Active[k] on line 3032, generated from ~Active[j] on line 3034, resets an entire column if entry is not active. Signal ~Active[k] is used in such situations as initializing a matrix or clearing an entry after the associated instruction graduates or aborts. Also shown is OR signal 3036 which represents row and column output values ORed together.

Fig. 31 shows timing for the dependency circuits.

C. Uncached Memory Dependency

Load and store instructions to uncached memory addresses are executed strictly in program order.

Processor 100 does not check for dependencies between cached and uncached accesses. This is not an issue in unmapped regions, because the cached and uncached address regions are disjoint. For mapped regions, however, TLB 422 may select different cache attributes for the same page. Processor 100 does not prevent the programmer from alternatively accessing the same memory address as "cached" and "uncached." However, coherency is not guaranteed; the contents of the cache are not checked for any uncached address.

While the above is a complete description of the preferred embodiment of the invention, various modifications, alternatives and equivalents may be used. Therefore, the above description should not be taken as limiting the scope of the invention which is defined by the appended claims.

What Is Claimed Is:

1. In an address queue for holding a plurality of entries used to access memory, a matrix of RAM cells comprising:

a first plurality of cells disposed in a row and associated with a first entry identifying a first set of entries which said first entry is dependent upon; and

a second plurality of cells disposed in a column and associated with said first entry identifying a second set of entries which are dependent upon said first entry.

2. An address queue for holding a plurality of entries used to access a set-associative data cache, said address queue comprising:

a comparator circuit for comparing a newly calculated partial address derived from a new queue entry with a previously calculated partial address derived from one of a plurality of previous entries;

a first matrix of RAM cells coupled to said comparator circuit for tracking all of said plurality of previous entries using a cache set that is also used by said new queue entry; and

a second matrix of RAM cells coupled to said comparator circuit for tracking entries that are store instructions which store a portion of data in said data cache which is accessed by a subsequent load instruction.

3. The address queue of claim 2 wherein said data cache comprises:

a first array for holding data, said first array being plurality-way interleaved having a first data way and a second data way;

a second array for holding a plurality of address tags associated with said data, said second array being plurality-way interleaved having a first tag way and a second tag way.

4. The address queue of claim 3 further comprising:

means for calculating a real address from one of said plurality of entries; and

means for comparing said real address with said plurality of address tags.

5. The address queue of claim 3 further comprising:

means for calculating a real address from one of said plurality of entries; and

means for comparing said real address with said plurality of address tags.

6. The address queue of claim 5 further comprising means for setting a use bit associated with said one of said plurality of entries when said real address matches an address tag held in said second tag way, said use bit indicating a sequential- order block within said cache set is accessed.

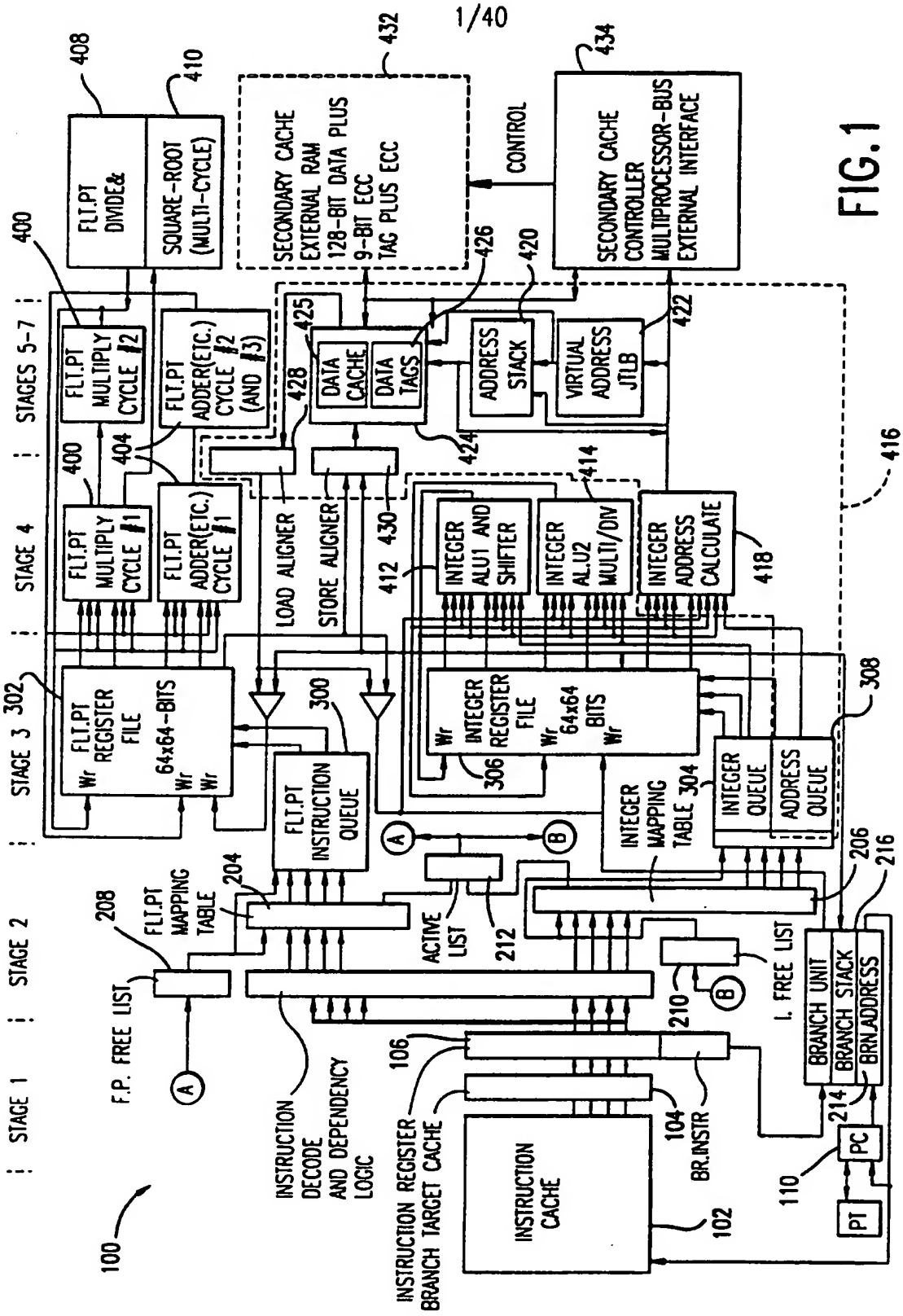

STAGE 1
STAGE 2
STAGE 3
STAGE 4
STAGES 5-7
100
F.P. FREE LIST
FLT.PT MAPPING TABLE
FLT.PT REGISTER FILE 64x64-BITS
FLT.PT INSTRUCTION QUEUE
INSTRUCTION DECODE AND DEPENDENCY LOGIC
INSTRUCTION REGISTER BRANCH TARGET CACHE
INSTRUCTION CACHE
BR.INSTR
I. FREE LIST
BRANCH UNIT
BRANCH STACK
BRN.ADDRESS
PC
PT
ACTIVE LIST
INTEGER MAPPING TABLE
INTEGER REGISTER FILE 64x64 BITS
INTEGER QUEUE
ADDRESS QUEUE
LOAD ALIGNER
STORE ALIGNER
INTEGER ALU1 AND SHIFTER
INTEGER ALU2 MULTI/DIV
INTEGER ADDRESS CALCULATE
FLT.PT MULTIPLY CYCLE #1
FLT.PT ADDER(ETC.) CYCLE #1
FLT.PT MULTIPLY CYCLE #2
FLT.PT ADDER(ETC.) CYCLE #2 (AND #3)
DATA CACHE
DATA TAGS
ADDRESS STACK
VIRTUAL ADDRESS JTLB
FLT.PT DIVIDE
SQUARE-ROOT (MULTI-CYCLE)
SECONDARY CACHE EXTERNAL RAM 128-BIT DATA PLUS 9-BIT ECC TAG PLUS ECC
CONTROL
SECONDARY CACHE CONTROLLER MULTIPROCESSOR-BUS EXTERNAL INTERFACE

FIG. 1

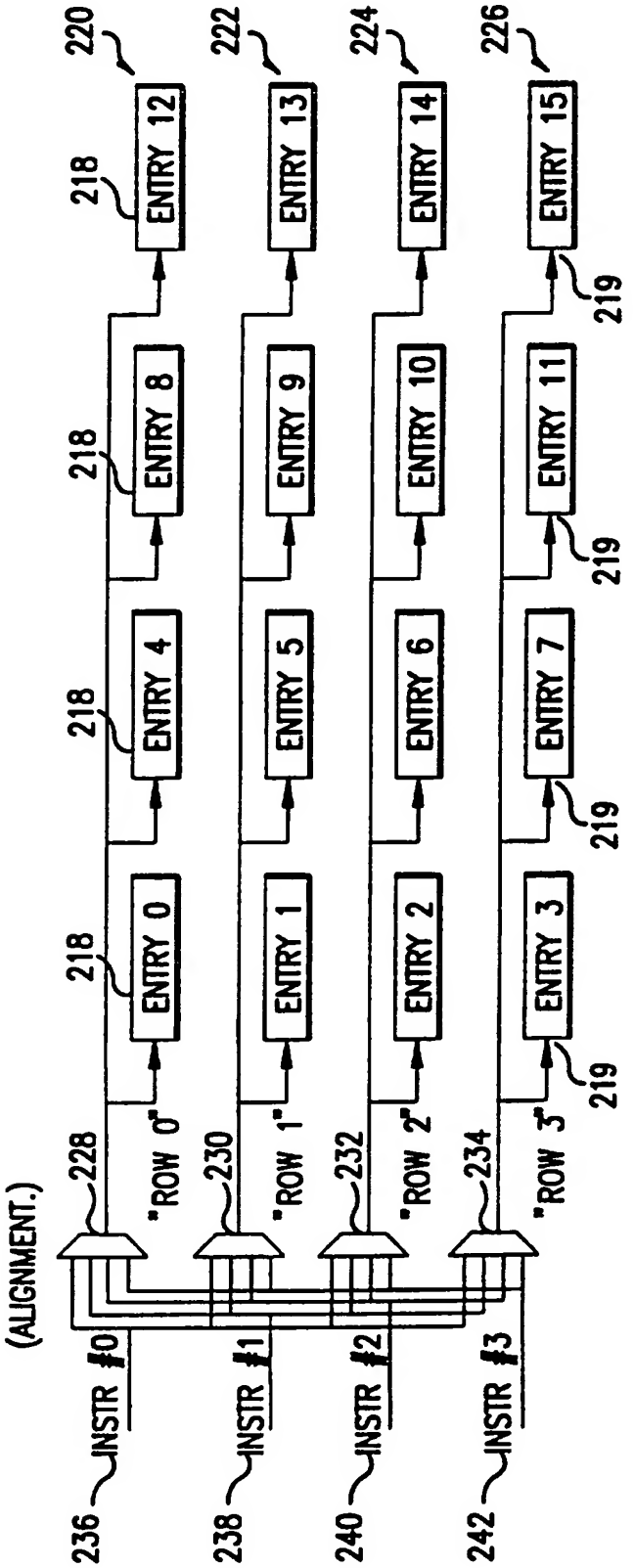

(ALIGNMENT.)
INSTR #0
INSTR #1
INSTR #2
INSTR #3
236
238
240
242
228
230
232
234
"ROW 0"
"ROW 1"
"ROW 2"
"ROW 3"
218
219
220
222
224
226
ENTRY 0
ENTRY 1
ENTRY 2
ENTRY 3
ENTRY 4
ENTRY 5
ENTRY 6
ENTRY 7
ENTRY 8
ENTRY 9
ENTRY 10
ENTRY 11
ENTRY 12
ENTRY 13
ENTRY 14
ENTRY 15

FIG.2

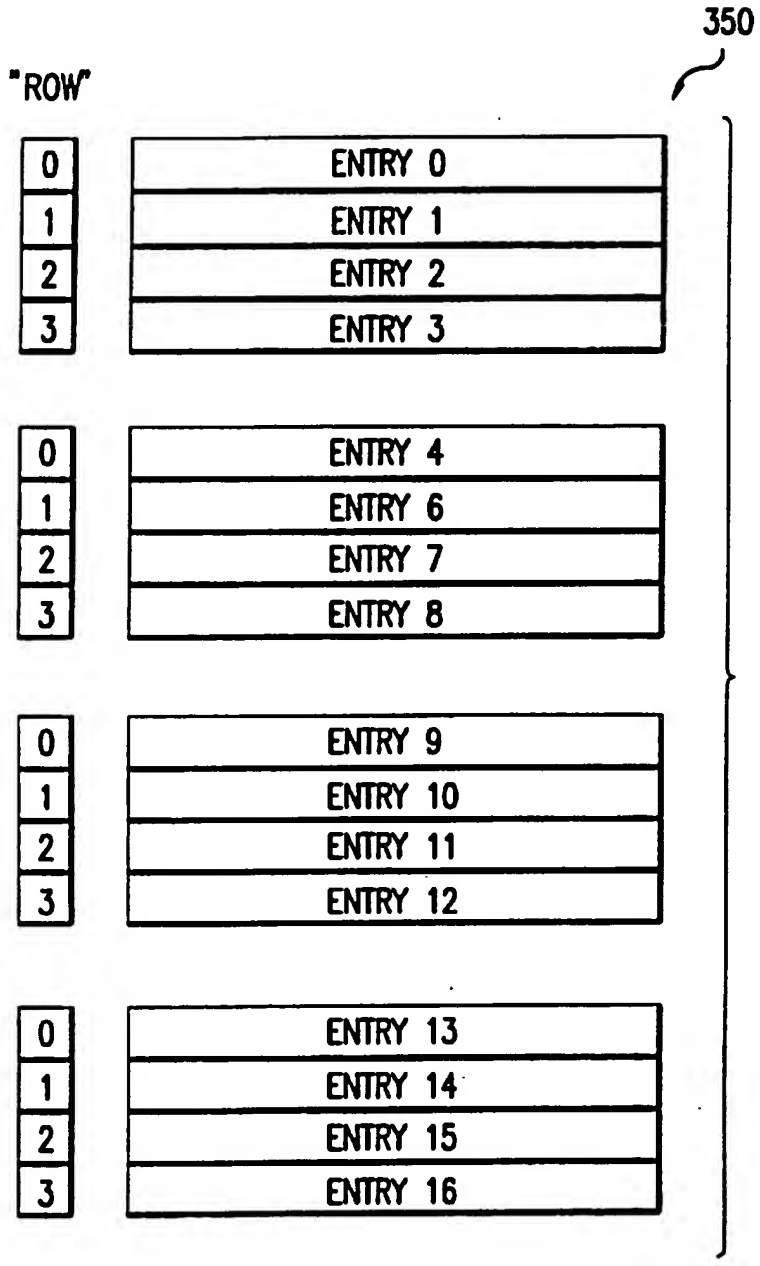

350
"ROW"
0
1
2
3
ENTRY 0
ENTRY 1
ENTRY 2
ENTRY 3
0
1
2
3
ENTRY 4
ENTRY 6
ENTRY 7
ENTRY 8
0
1
2
3
ENTRY 9
ENTRY 10
ENTRY 11
ENTRY 12
0
1
2
3
ENTRY 13
ENTRY 14
ENTRY 15
ENTRY 16

FIG.3

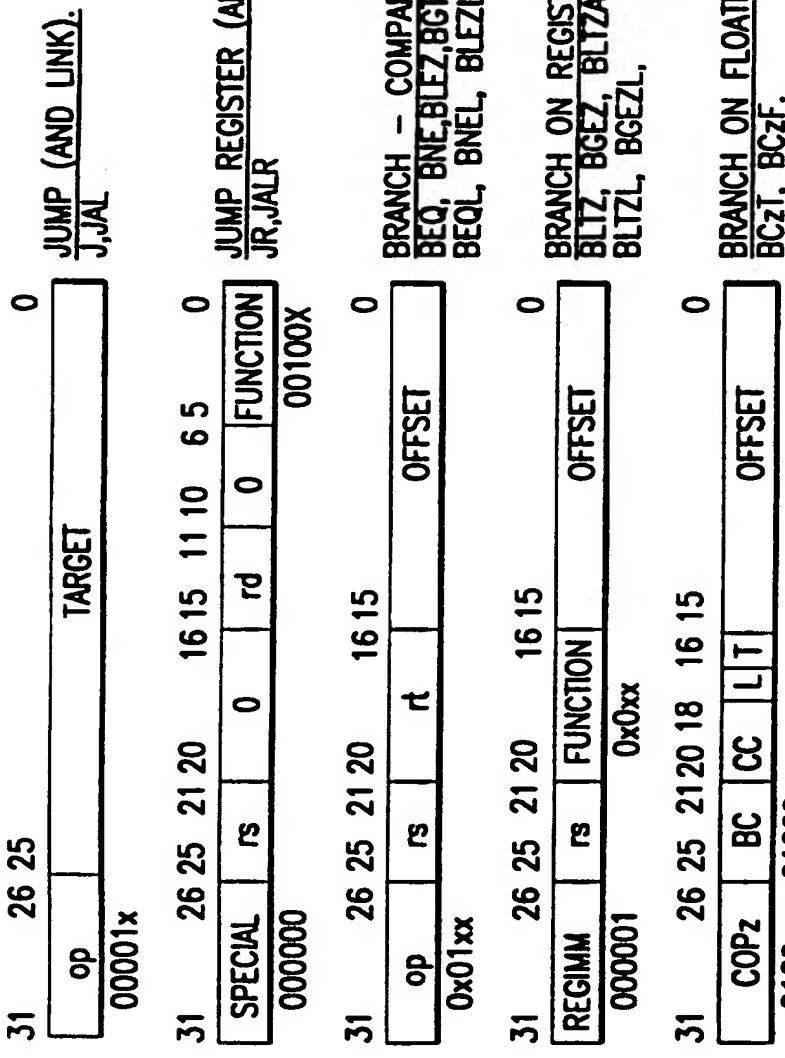

31 26 25 0
op
TARGET
00001x
JUMP (AND LINK).
J,JAL
31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL
rs
0
rd
0
FUNCTION
000000
00100X
JUMP REGISTER (AND LINK).
JR,JALR
31 26 25 21 20 16 15 0
op
rs
rt
OFFSET
0x01xx
BRANCH - COMPARE TWO REGISTERS.
BEQ, BNE,BLEZ,BGTZ
BEQL, BNEL, BLEZL, BGTZL
31 26 25 21 20 16 15 0
REGIMM
rs
FUNCTION
OFFSET
000001
0x0xx
BRANCH ON REGISTER VALUE.
BLTZ, BGEZ, BLTZAL, BGEZAL
BLTZL, BGEZL,
31 26 25 21 20 18 16 15 0
COPz
BC
CC
L
T
OFFSET
0100xx 01000
BRANCH ON FLOATING-PT CONDITION
BCzT, BCzF.
BCzTL, BCzFL
(FLOATING-POINT CONDITION CODE SELECT CC IS USED ONLY IN MIPS-4 ARCHITECTURE. L='LIKELY'. T='TRUE'.)

FIG.4

FIG.5A

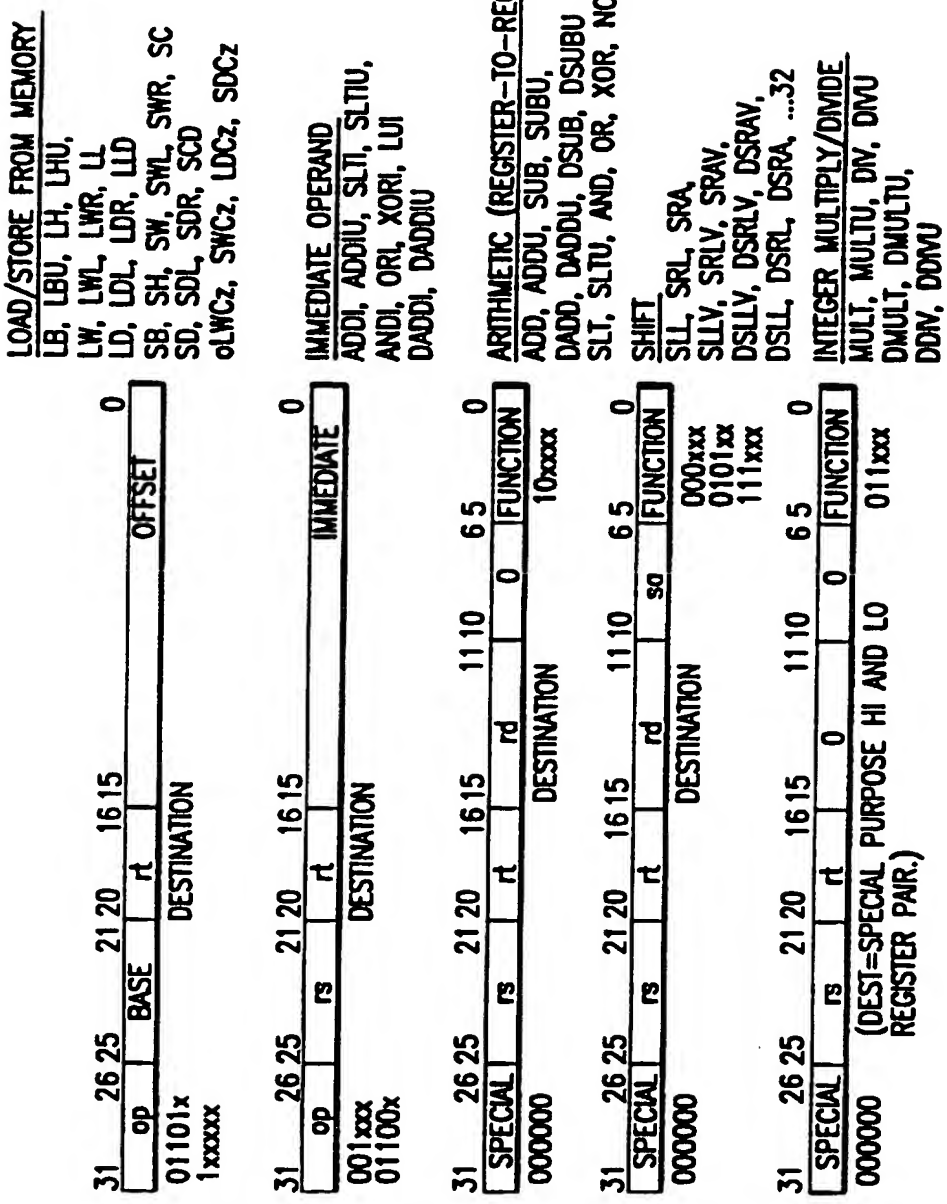

LOAD/STORE FROM MEMORY
LB, LBU, LH, LHU,
LW, LWL, LWR, LL
LD, LDL, LDR, LLD
SB, SH, SW, SWL, SWR, SC
SD, SDL, SDR, SCD
oLWCz, SWCz, LDCz, SDCz
31 26 25 21 20 16 15 0
op BASE rt OFFSET
01101x
1xxxxx
DESTINATION
IMMEDIATE OPERAND
ADDI, ADDIU, SLTI, SLTIU,
ANDI, ORI, XORI, LUI
DADDI, DADDIU
31 26 25 21 20 16 15 0
op rs rt IMMEDIATE
001xxx
01100x
DESTINATION
ARITHMETIC (REGISTER-TO-REGISTER)
ADD, ADDU, SUB, SUBU,
DADD, DADDU, DSUB, DSUBU
SLT, SLTU, AND, OR, XOR, NOR
31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL rs rt rd 0 FUNCTION
000000
10xxxx
DESTINATION
SHIFT
SLL, SRL, SRA,
SLLV, SRLV, SRAV,
DSLLV, DSRLV, DSRAV,
DSLL, DSRL, DSRA, ...32
31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL rs rt rd sa FUNCTION
000000
000xxx
0101xx
111xxx
DESTINATION
INTEGER MULTIPLY/DIVIDE
MULT, MULTU, DIV, DIVU
DMULT, DMULTU,
DDIV, DDIVU
31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL rs rt 0 0 FUNCTION
000000
011xxx
(DEST=SPECIAL PURPOSE HI AND LO
REGISTER PAIR.)

FIG.5B

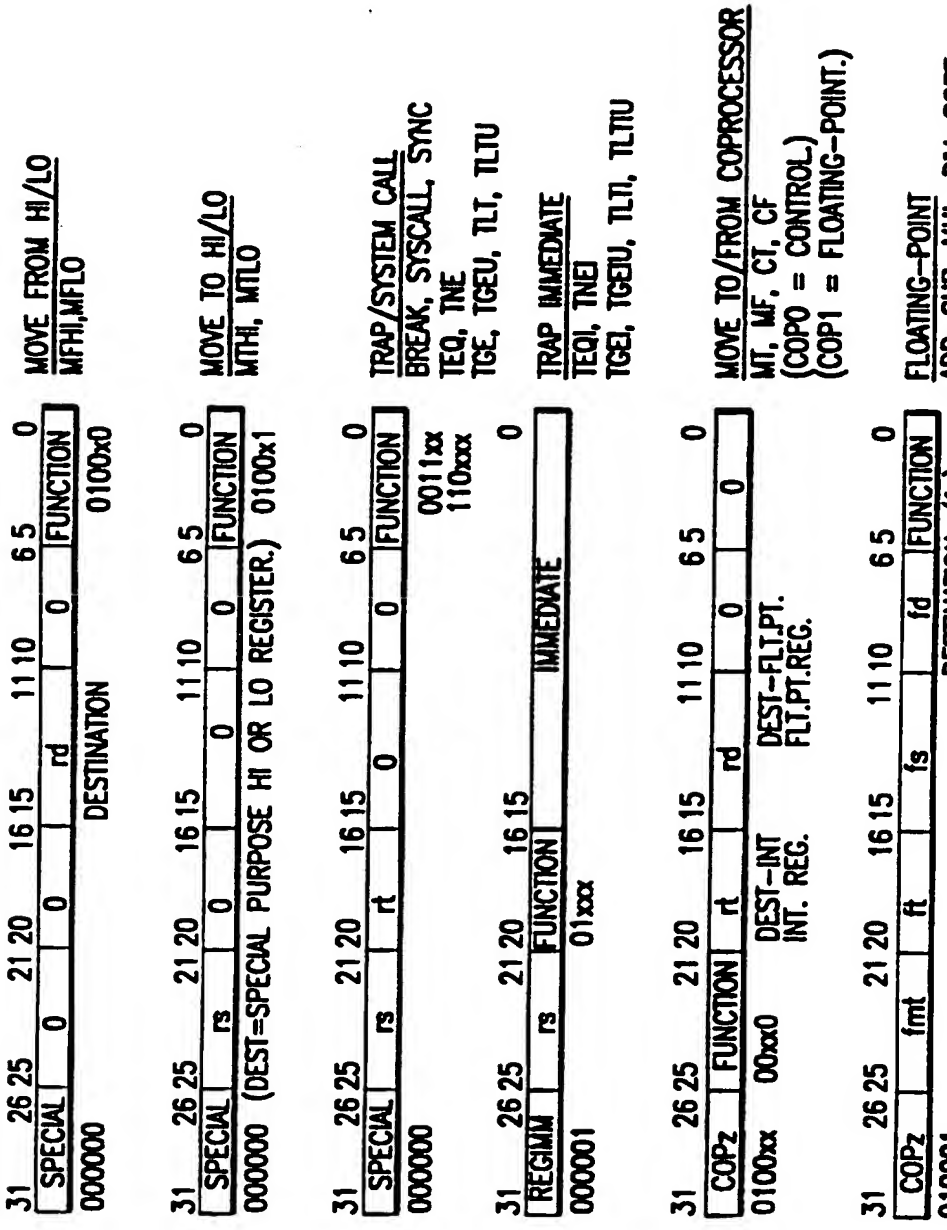
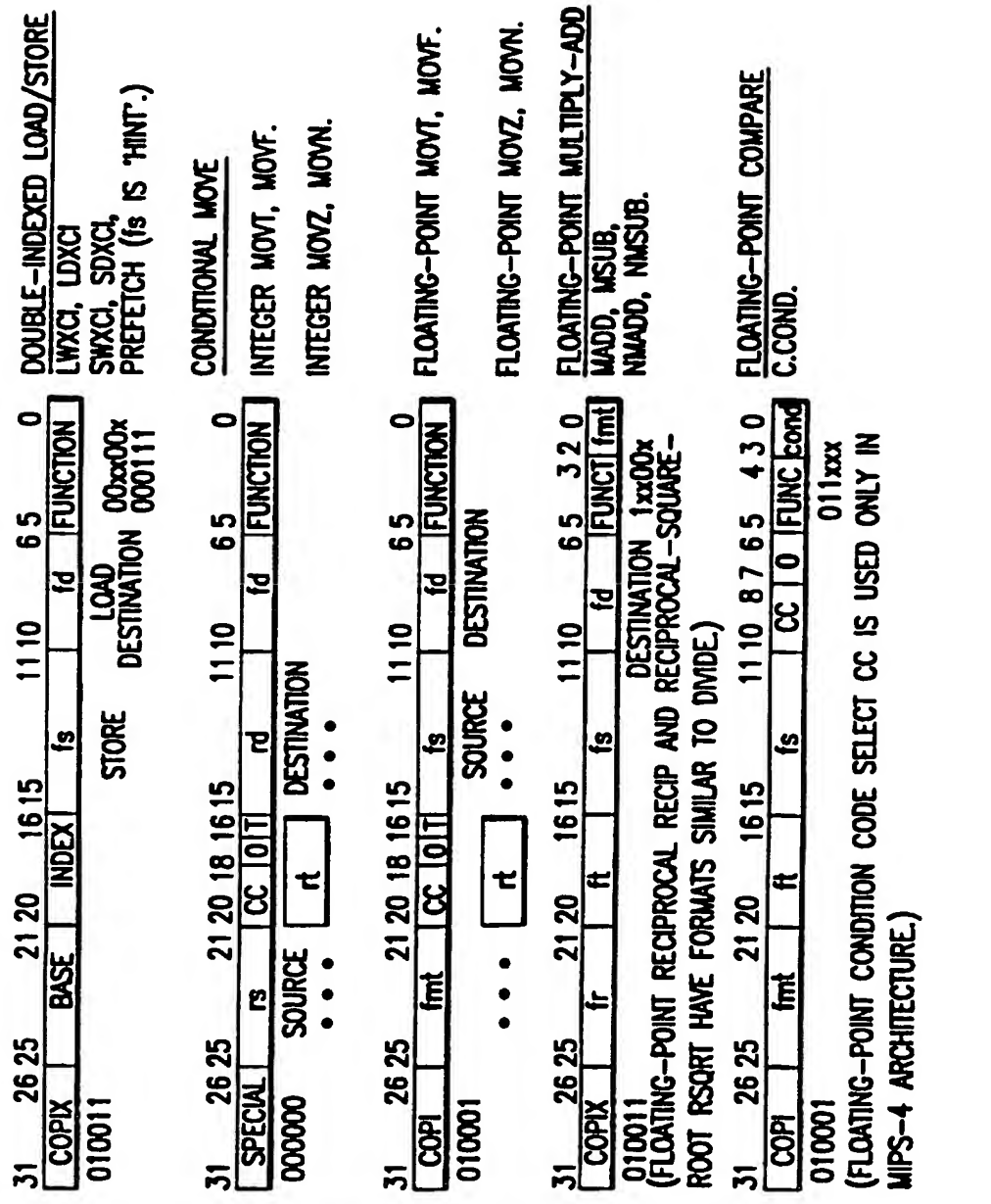

31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL 0 0 rd 0 FUNCTION
000000 DESTINATION 0100x0
MOVE FROM HI/LO
MFHI,MFLO
31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL rs 0 0 0 FUNCTION
000000 (DEST=SPECIAL PURPOSE HI OR LO REGISTER.) 0100x1
MOVE TO HI/LO
MTHI, MTLO
31 26 25 21 20 16 15 11 10 6 5 0
SPECIAL rs rt 0 0 FUNCTION
000000 0011xx 110xxx
TRAP/SYSTEM CALL
BREAK, SYSCALL, SYNC
TEQ, TNE
TGE, TGEU, TLT, TLTU
31 26 25 21 20 16 15 0
REGIMM rs FUNCTION IMMEDIATE
000001 01xxx
TRAP IMMEDIATE
TEQI, TNEI
TGEI, TGEIU, TLTI, TLTIU
31 26 25 21 20 16 15 11 10 6 5 0
COPz FUNCTION rt rd 0 0
0100xx 00xx0 DEST-INT INT. REG. DEST-FLT.PT. FLT.PT.REG.
MOVE TO/FROM COPROCESSOR
MT, MF, CT, CF
(COP0 = CONTROL.)
(COP1 = FLOATING-POINT.)
31 26 25 21 20 16 15 11 10 6 5 0
COPz fmt ft fs fd FUNCTION
010001 DESTINATION (fd)
FLOATING-POINT
ADD, SUB, MUL, DIV, SQRT.
ABS, MOV, NEG, CVT ...

FIG.6


31 26 25 21 20 16 15 11 10 6 5 0
COP1X | BASE | INDEX | fs | fd | FUNCTION
010011
STORE LOAD
DESTINATION 00xx00x
000111
DOUBLE-INDEXED LOAD/STORE
LWXC1, LDXC1
SWXC1, SDXC1,
PREFETCH (fs IS "HINT.")
31 26 25 21 20 18 16 15 11 10 6 5 0
SPECIAL | rs | CC | 0 | T | rd | fd | FUNCTION
000000 SOURCE rt DESTINATION
CONDITIONAL MOVE
INTEGER MOVT, MOVF.
INTEGER MOVZ, MOVN.
31 26 25 21 20 18 16 15 11 10 6 5 0
COP1 | fmt | CC | 0 | T | fs | fd | FUNCTION
010001 SOURCE rt DESTINATION
FLOATING-POINT MOVT, MOVF.
FLOATING-POINT MOVZ, MOVN.
31 26 25 21 20 16 15 11 10 6 5 3 2 0
COP1X | fr | ft | fs | fd | FUNCT | fmt
010011 DESTINATION 1xx00x
(FLOATING-POINT RECIPROCAL RECIP AND RECIPROCAL-SQUARE-
ROOT RSQRT HAVE FORMATS SIMILAR TO DIVIDE.)
FLOATING-POINT MULTIPLY-ADD
MADD, MSUB,
NMADD, NMSUB.
31 26 25 21 20 16 15 11 10 8 7 6 5 4 3 0
COP1 | fmt | ft | fs | CC | 0 | FUNC | cond
010001 011xxx
(FLOATING-POINT CONDITION CODE SELECT CC IS USED ONLY IN
MIPS-4 ARCHITECTURE.)
FLOATING-POINT COMPARE
C.COND.

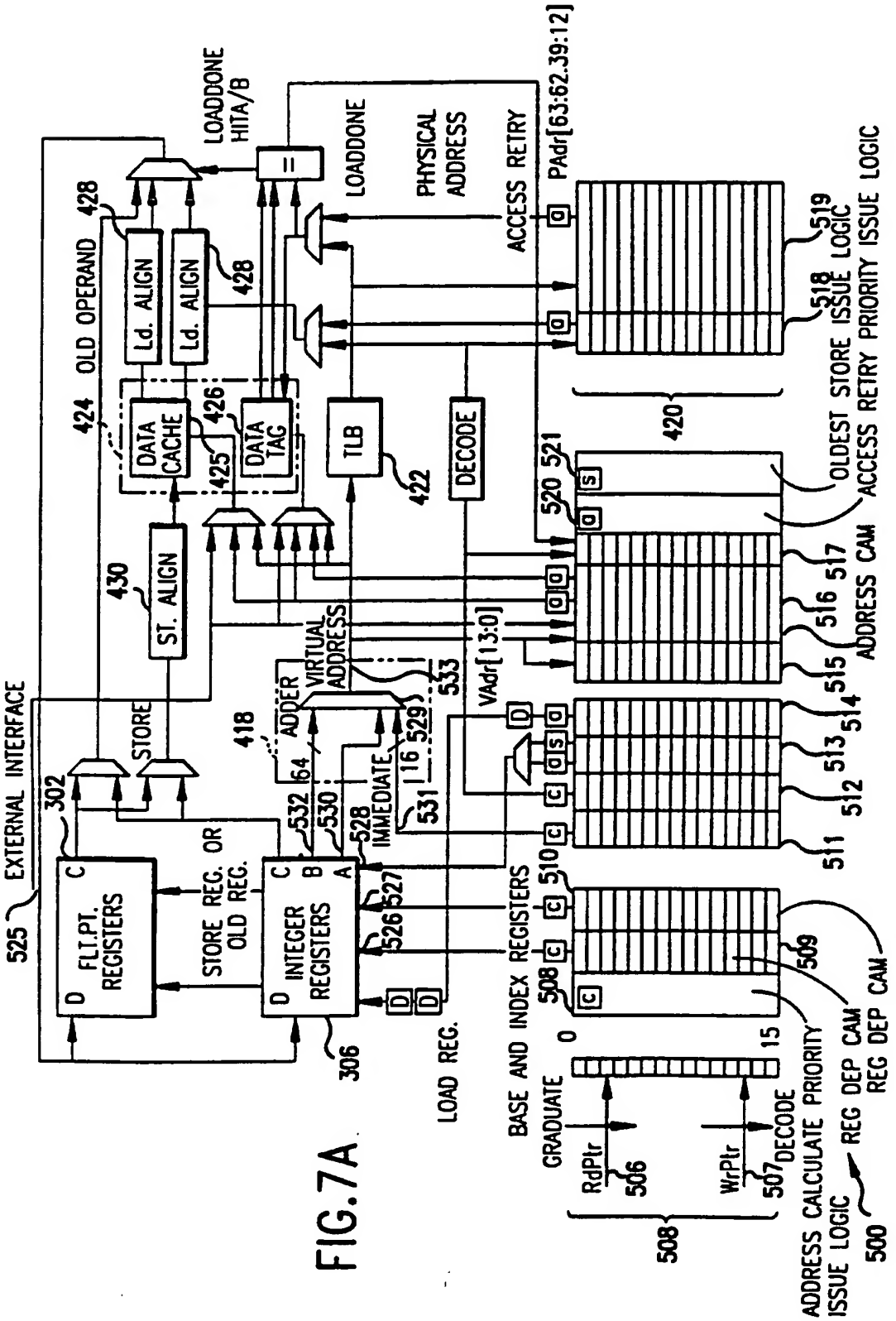

EXTERNAL INTERFACE
525
302
FLT.PT. REGISTERS
INTEGER REGISTERS
306
STORE
STORE REG. OR OLD REG.
LOAD REG.
418
ADDER
VIRTUAL ADDRESS
533
VAdr[13:0]
IMMEDIATE
64
16
528
529
530
531
532
527
526
LOADDONE HITA/B
LOADDONE
PHYSICAL ADDRESS
ACCESS RETRY
PAdr[63:62,39:12]
428
OLD OPERAND
424
Ld. ALIGN
Ld. ALIGN
DATA CACHE
425
426
DATA TAG
TLB
422
DECODE
430
ST. ALIGN
420
519
518
OLDEST STORE ISSUE LOGIC
ACCESS RETRY PRIORITY ISSUE LOGIC
520
521
517
516
515
514
513
512
511
ADDRESS CAM
510
509
508
BASE AND INDEX REGISTERS
GRADUATE
RdPtr
506
WrPtr
507
DECODE
508
0
15
ADDRESS CALCULATE PRIORITY ISSUE LOGIC
500
REG DEP CAM
REG DEP CAM

FIG.7A

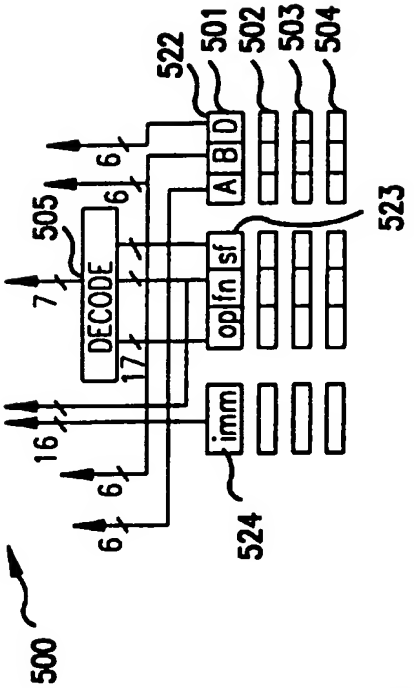
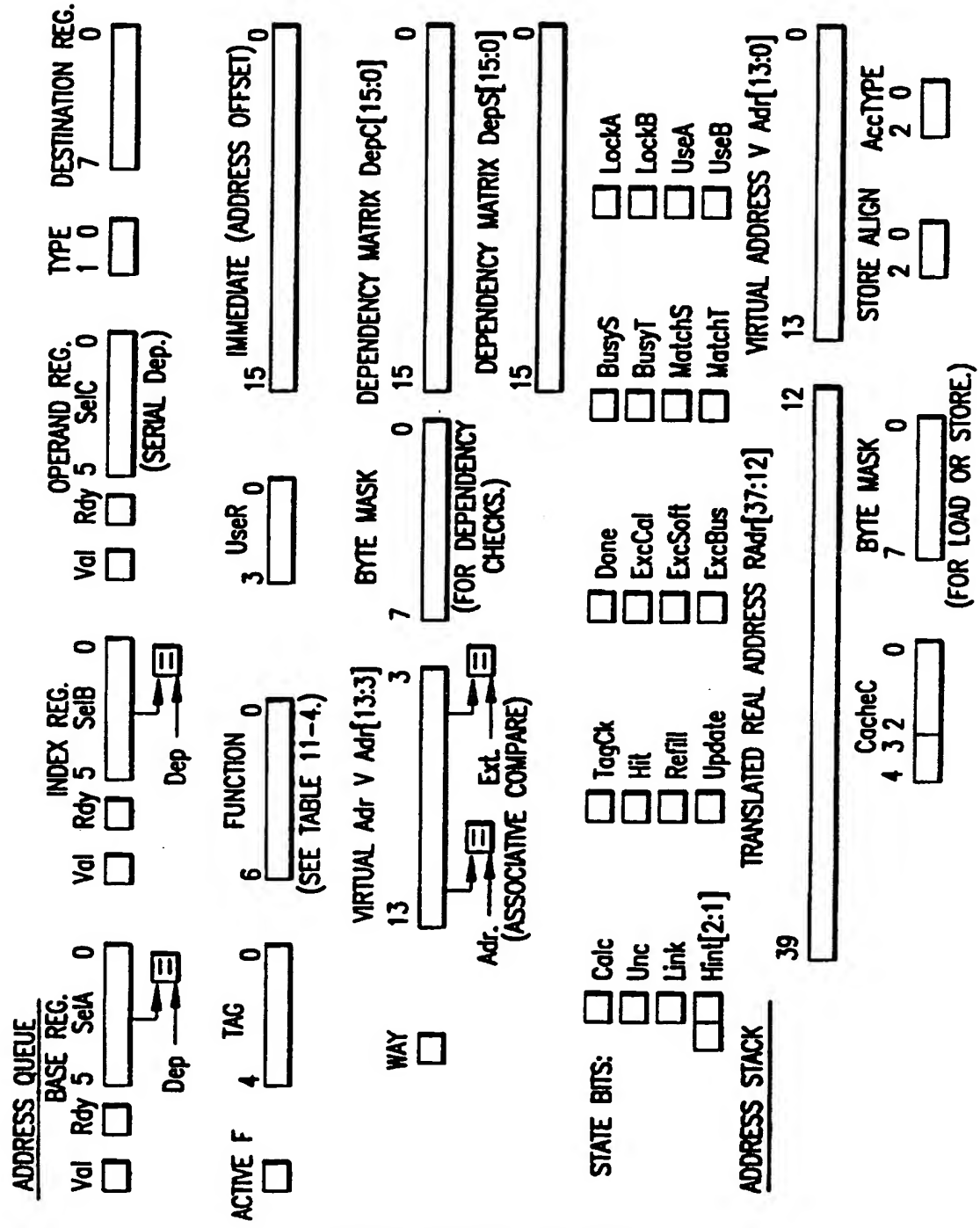

522
501
502
503
504
A B D
6
6
505
DECODE
7
17
523
opfn sf
16
6
6
imm
524
500

FIG.7B

INSTRUCTION	OPERAND A (BASE REG.)	OPERAND B (INDEX REG.) (INT.STORE)	OPERAND C (FLT.STORE)	IMMEDIATE (OFFSET)	DESTINATION REGISTER	FUNCTION (MODIFIED) OPCODE INSTR[31:26] FUNCTION INSTR[5:0] SUBFUNCTION INSTR[20:16]
LOAD (LEFT/RIGHT) LPF PREFETCH	V rs	0 – V rd (OLD)	0 – 0 rd (OLD)	INSTR [15:6]	DO NEW rt Oh (HINT)	nnnnnn aaaaaa xxxxx 100x10 aaaaaa xxxxx 111011 aaaaaa xxxxx
LOAD FLT.PT. (MIPS-2).[2]	V rs	0 – FLTHI=HIGH HALF	0 – 1 fd (OLD)	INSTR[15:6]	DO NEW ft	110n01 aaaaaa xxxxx
LOAD INDEX[1] (MIPS-2).[2] PFETCH	V rs	V rt FLTHI=HIGH HALF	0 – 1 fd (OLD)	(UNUSED)	11 NEW fd Oh (HINT)	011011 000nnn xxxxx 011011 000111 xxxxx
CACHE OP	V rs	0 –	0 –	INSTR[15:6]	00 –	101111 aaaaaa nnn cc
STORE	V rs	V rt	0 rt	INSTR[15:6]	00 –	nnnnnn aaaaaa xxxxx
STORE COND.	V rs	V rt	0 rt	INSTR[15:6]	DO NEW rt	000n00 aaaaaa xxxxx
STORE FLT.PT (MIPS-2).[2]	V rs	0 – FLTHI=HIGH HALF	1 ft	INSTR[15:6]	00 –	111n01 aaaaaa xxxxx
STORE INDEX[1] (MIPS-2).[2]	V rs	V rt FLTHI=HIGH HALF	1 fd	(UNUSED)	00 –	101111 001nnn xxxxx

FIG.8

FIG.9


ADDRESS QUEUE
BASE REG.
Val Rdy 5 SelA 0
Dep
ACTIVE F
TAG 4 0
INDEX REG.
Val Rdy 5 SelB 0
Dep
FUNCTION 6 0
(SEE TABLE 11-4.)
WAY
VIRTUAL Adr V Adr[13:3] 13 3
Adr.
Ext.
(ASSOCIATIVE COMPARE)
STATE BITS:
Calc
Unc
Link
Hint[2:1]
TagCk
Hit
Refill
Update
Done
ExcCal
ExcSoft
ExcBus
OPERAND REG.
Val Rdy 5 SelC 0
(SERIAL Dep.)
UseR 3 0
BYTE MASK 7 0
(FOR DEPENDENCY CHECKS.)
TRANSLATED REAL ADDRESS RAdr[37:12] 39 12
CacheC 4 3 2 0
BYTE MASK 7 0
(FOR LOAD OR STORE.)
TYPE 1 0
IMMEDIATE (ADDRESS OFFSET) 15 0
DEPENDENCY MATRIX DepC[15:0] 15 0
DEPENDENCY MATRIX DepS[15:0] 15 0
BusyS
BusyT
MatchS
MatchT
LockA
LockB
UseA
UseB
VIRTUAL ADDRESS V Adr[13:0] 13 0
STORE ALIGN 2 0
DESTINATION REG. 7 0
AccTYPE 2 0
ADDRESS STACK

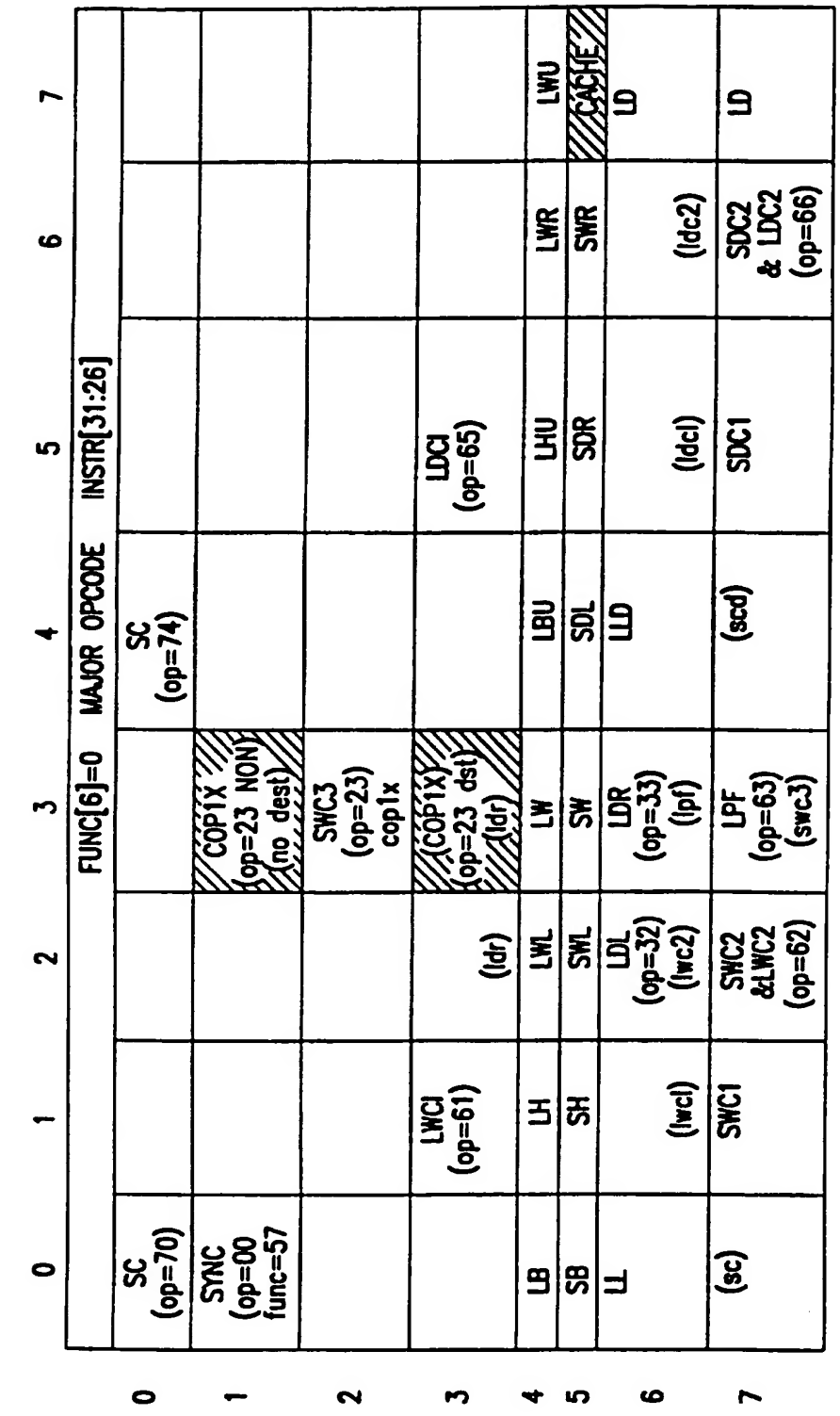

FUNC[6]=0 MAJOR OPCODE INSTR[31:26]
0 1 2 3 4 5 6 7
SC (op=70)
SYNC (op=00 func=57
LB
SB
LL
(sc)
LWCI (op=61)
LH
SH
(lwcl)
SWC1
(ldr)
LWL
SWL
LDL (op=32) (lwc2)
SWC2 &LWC2 (op=62)
COP1X (op=23 NONE no dest)
LW
SW
LDR (op=33) (lpf)
SWC3 (op=23) cop1x
(COP1X) (op=23 dst) (ldr)
LDCI (op=65)
SC (op=74)
LBU
SDL
LLD
(scd)
LHU
SDR
(ldcl)
SDC1
LWR
SWR
(ldc2)
SDC2 & LDC2 (op=66)
LWU
CACHE
LD
LD
0
1
2
3
4
5
6
7

FIG.10A

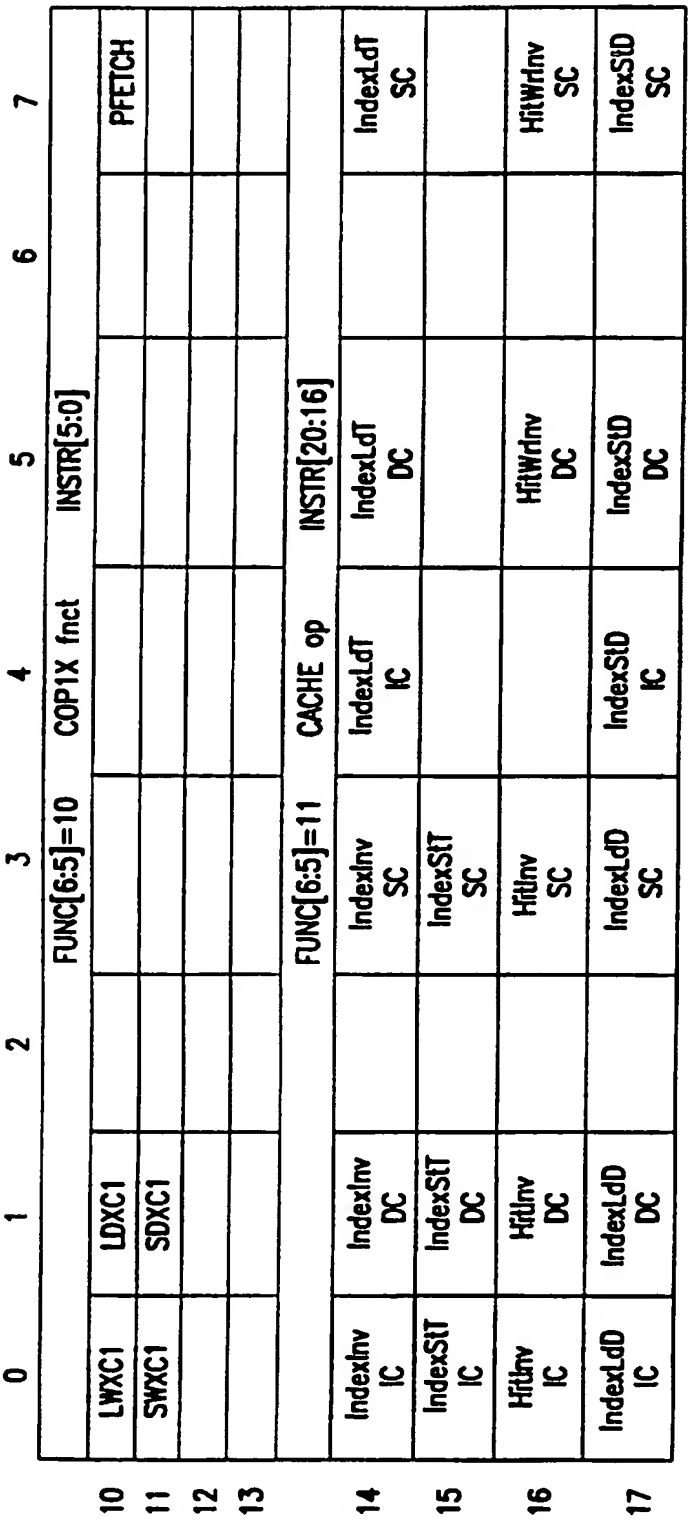

FUNC[6:5]=10 COP1X fnct INSTR[5:0]
FUNC[6:5]=11 CACHE op INSTR[20:16]
0 1 2 3 4 5 6 7
10 11 12 13 14 15 16 17
LWXC1 LDXC1 PFETCH
SWXC1 SDXC1
IndexInv IC
IndexInv DC
IndexInv SC
IndexLdT IC
IndexLdT DC
IndexLdT SC
IndexStT IC
IndexStT DC
IndexStT SC
HitInv IC
HitInv DC
HitInv SC
HitWrInv DC
HitWrInv SC
IndexLdD IC
IndexLdD DC
IndexLdD SC
IndexStD IC
IndexStD DC
IndexStD SC

FIG.10B

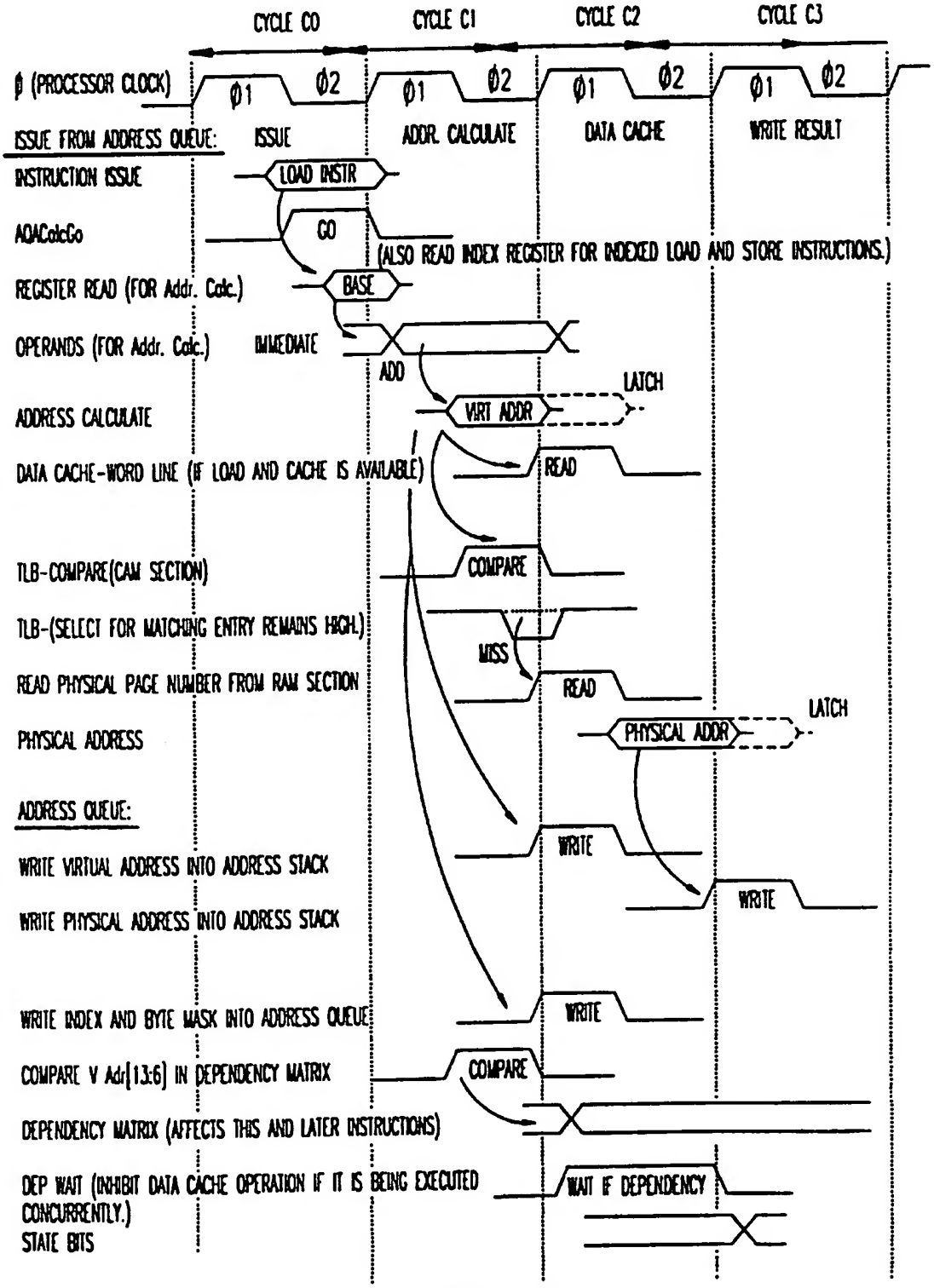

CYCLE C0
CYCLE C1
CYCLE C2
CYCLE C3
Φ (PROCESSOR CLOCK)
Φ1
Φ2
ISSUE FROM ADDRESS QUEUE:
ISSUE
ADDR. CALCULATE
DATA CACHE
WRITE RESULT
INSTRUCTION ISSUE
LOAD INSTR
AOACalcGo
GO
(ALSO READ INDEX REGISTER FOR INDEXED LOAD AND STORE INSTRUCTIONS.)
REGISTER READ (FOR Addr. Calc.)
BASE
OPERANDS (FOR Addr. Calc.)
IMMEDIATE
ADD
LATCH
ADDRESS CALCULATE
VIRT ADDR
DATA CACHE-WORD LINE (IF LOAD AND CACHE IS AVAILABLE)
READ
TLB-COMPARE(CAM SECTION)
COMPARE
TLB-(SELECT FOR MATCHING ENTRY REMAINS HIGH.)
MISS
READ PHYSICAL PAGE NUMBER FROM RAM SECTION
READ
LATCH
PHYSICAL ADDRESS
PHYSICAL ADDR
ADDRESS QUEUE:
WRITE VIRTUAL ADDRESS INTO ADDRESS STACK
WRITE
WRITE PHYSICAL ADDRESS INTO ADDRESS STACK
WRITE
WRITE INDEX AND BYTE MASK INTO ADDRESS QUEUE
WRITE
COMPARE V Adr[13:6] IN DEPENDENCY MATRIX
COMPARE
DEPENDENCY MATRIX (AFFECTS THIS AND LATER INSTRUCTIONS)
DEP WAIT (INHIBIT DATA CACHE OPERATION IF IT IS BEING EXECUTED CONCURRENTLY.)
WAIT IF DEPENDENCY
STATE BITS

FIG.11

EACH REQUEST IS FOR ONE BANK OF DATA CACHE ONLY:	DATA CACHE ARRAYS				PROCESSOR UNITS			REGISTER FILES	
	BANK 1		BANK 0						
CYCLE "C0"	TAG	DATA	TAG	DATA	TAGCK	LOAD	STORE	READ	WRITE
1. EXTERNAL INTERFACE (DECODE EXODCmd). (EXOAdr[5] SELECTS EITHER BANK.)	▨	▨	▨	▨	[ ]	▧ "FREELOAD"	[ ]	▧ MODIFY	▧ "FREELOAD"
2. OLDEST STORE INSTRUCTION, IF GUARANTEED TO GRADUATE.	[ ]	▨	[ ]	▨	[ ]	[ ]	▨	▨	[ ]
3. RETRY FROM ADDRESS QUEUE. TAG (& MAYBE DATA).	▨	▧ LOAD	▨	▧ LOAD	▨	▧ LOAD	[ ]	▧ MODIFY	▧ LOAD
DATA ONLY.	[ ]	▨	[ ]	▨	[ ]	▨ "JUSTLOAD"	[ ]	▧ MODIFY	▨ "JUSTLOAD"
CYCLE "C1"									
4. OLDEST STORE INSTRUCTION, IF IT MIGHT GRADUATE.	[ ]	▨	[ ]	▨	[ ]	[ ]	▨	▨	[ ]
5. NEW INSTRUCTION FROM ADDRESS CALCULATE. (VAdr[5] SELECTS EITHER BANK.)	▨	▧ LOAD	▨	▧ LOAD	▨	▧	[ ]	▨	▧

LEGEND: CACHE ARRAYS: EACH REQUESTOR ACCESS ONLY ONE BANK OF THE DATA CACHE.

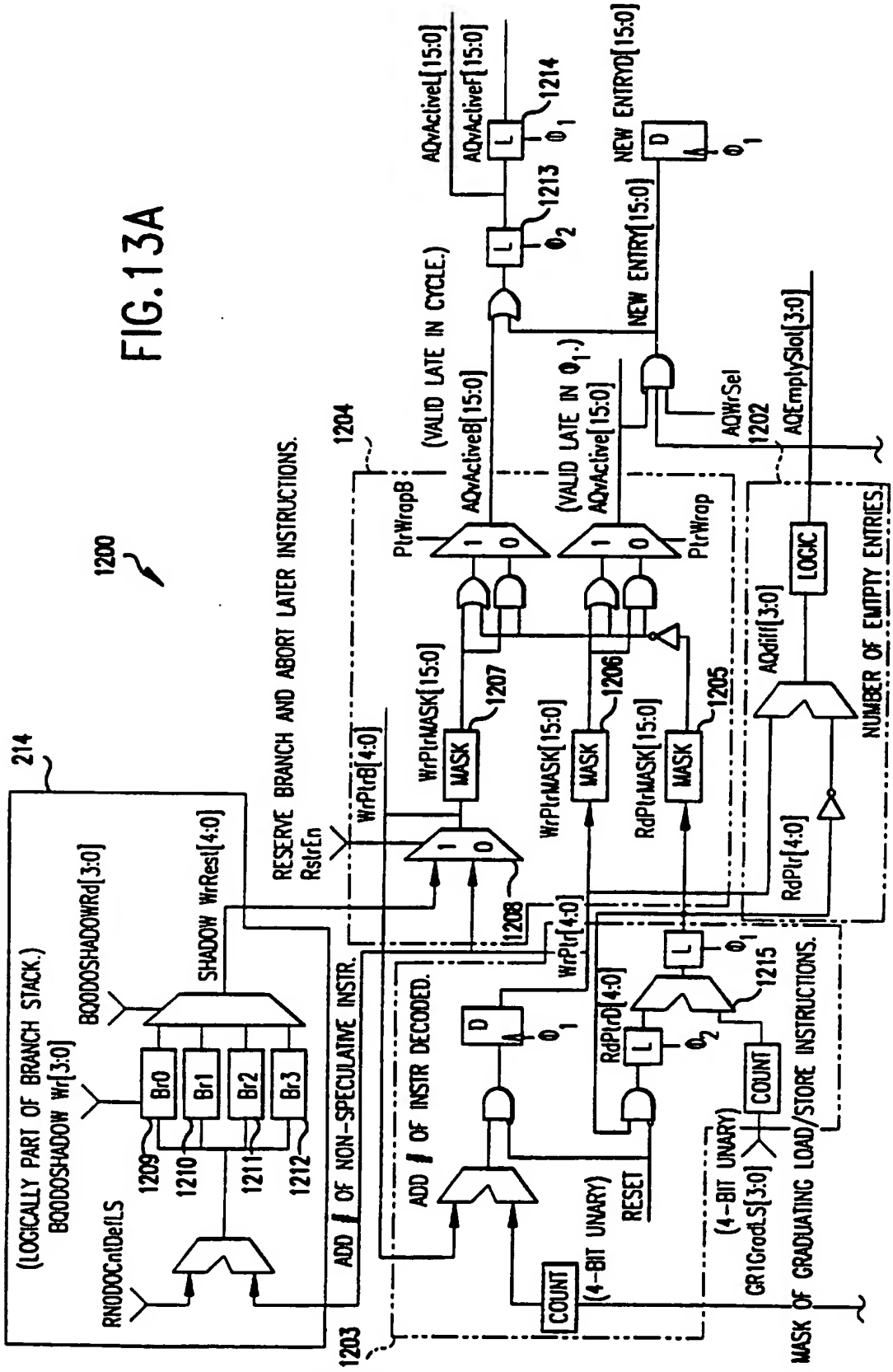
▨ THIS ARRAY OR RESOURCE IS REQUIRED. (EXTERNAL INTERFACE SPECIFIES WHICH ARRAYS IT NEEDS.)

▧ THIS ARRAY OR RESOURCE IS REQUIRED ONLY IN SPECIAL CASES.

[ ] THIS ARRAY OR RESOURCE IS NOT USED.

FIG.12

FIG. 13A


1200
214
(LOGICALLY PART OF BRANCH STACK.)
BOODOSHADOW Wr[3:0]
BOODOSHADOWRd[3:0]
SHADOW WrRes[4:0]
RNOOOCntDefLS
Br0
Br1
Br2
Br3
1209
1210
1211
1212
1203
ADD # OF NON-SPECULATIVE INSTR.
RESERVE BRANCH AND ABORT LATER INSTRUCTIONS.
RstrEn
WrPtrB[4:0]
1204
PtrWrapB
(VALID LATE IN CYCLE.)
AQvActiveB[15:0]
AQvActiveL[15:0]
AQvActiveF[15:0]
1214
1213
NEW ENTRY0[15:0]
NEW ENTRY[15:0]
(VALID LATE IN ϕ_1.)
AQvActive[15:0]
PtrWrap
AQWrSel
1202
AQEmptySlot[3:0]
WrPtrMASK[15:0]
1207
WrPtrMASK[15:0]
1206
RdPtrMASK[15:0]
1205
MASK
LOGIC
AQdiff[3:0]
NUMBER OF EMPTY ENTRIES.
1208
WrPtr[4:0]
RdPtr[4:0]
1215
ADD # OF INSTR DECODED.
RdPtrD[4:0]
COUNT
(4-BIT UNARY)
RESET
(4-BIT UNARY)
GR1GradLS[3:0]
MASK OF GRADUATING LOAD/STORE INSTRUCTIONS.

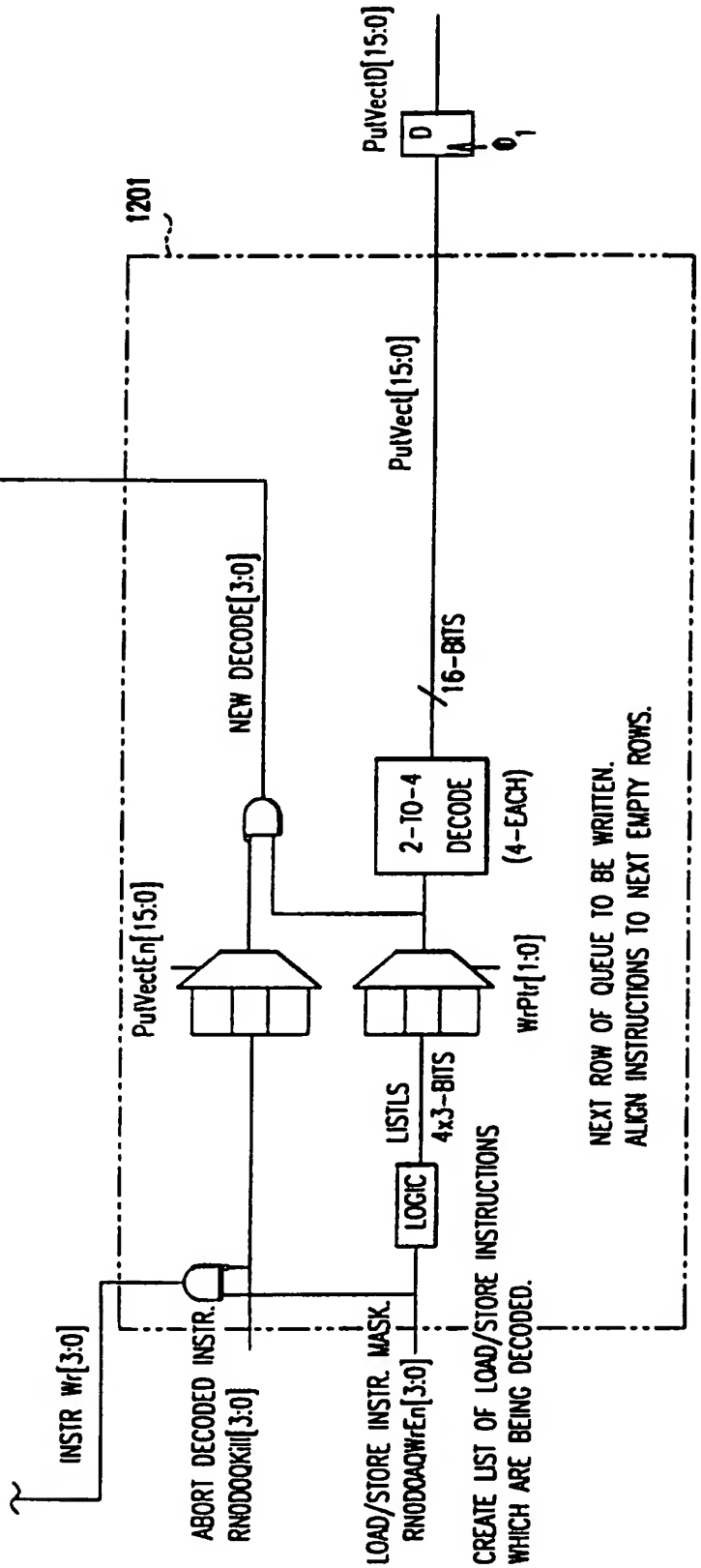

PutVectD[15:0]
D
Q
1201
PutVect[15:0]
NEW DECODE[3:0]
16-BITS
2-TO-4 DECODE
(4-EACH)
PutVectEn[15:0]
WrPtr[1:0]
LISTLS
4x3-BITS
LOGIC
INSTR Wr[3:0]
ABORT DECODED INSTR. RNODDOOKill[3:0]
LOAD/STORE INSTR. MASK. RNODDAQWrEn[3:0]
CREATE LIST OF LOAD/STORE INSTRUCTIONS WHICH ARE BEING DECODED.
NEXT ROW OF QUEUE TO BE WRITTEN. ALIGN INSTRUCTIONS TO NEXT EMPTY ROWS.

FIG.13B

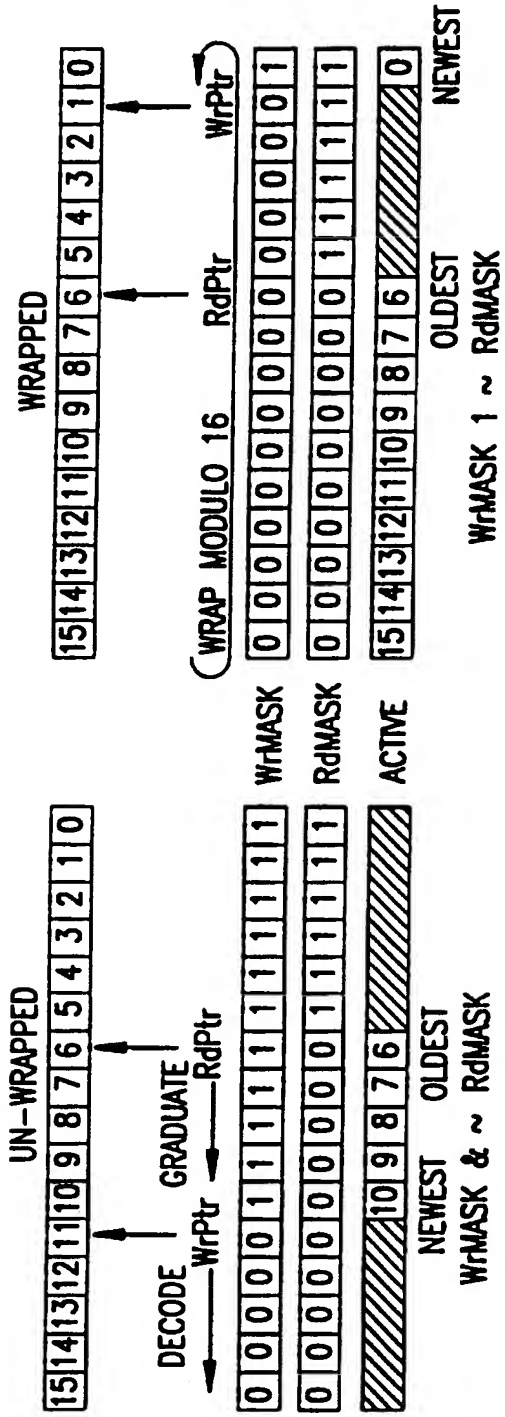

WRAPPED
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
WRAP MODULO 16
RdPtr
WrPtr
WrMASK
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
RdMASK
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
ACTIVE
15 14 13 12 11 10 9 8 7 6
OLDEST
NEWEST
WrMASK 1 ~ RdMASK
UN-WRAPPED
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
DECODE
WrPtr
GRADUATE
RdPtr
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
10 9 8 7 6
NEWEST
OLDEST
WrMASK & ~ RdMASK

FIG.14

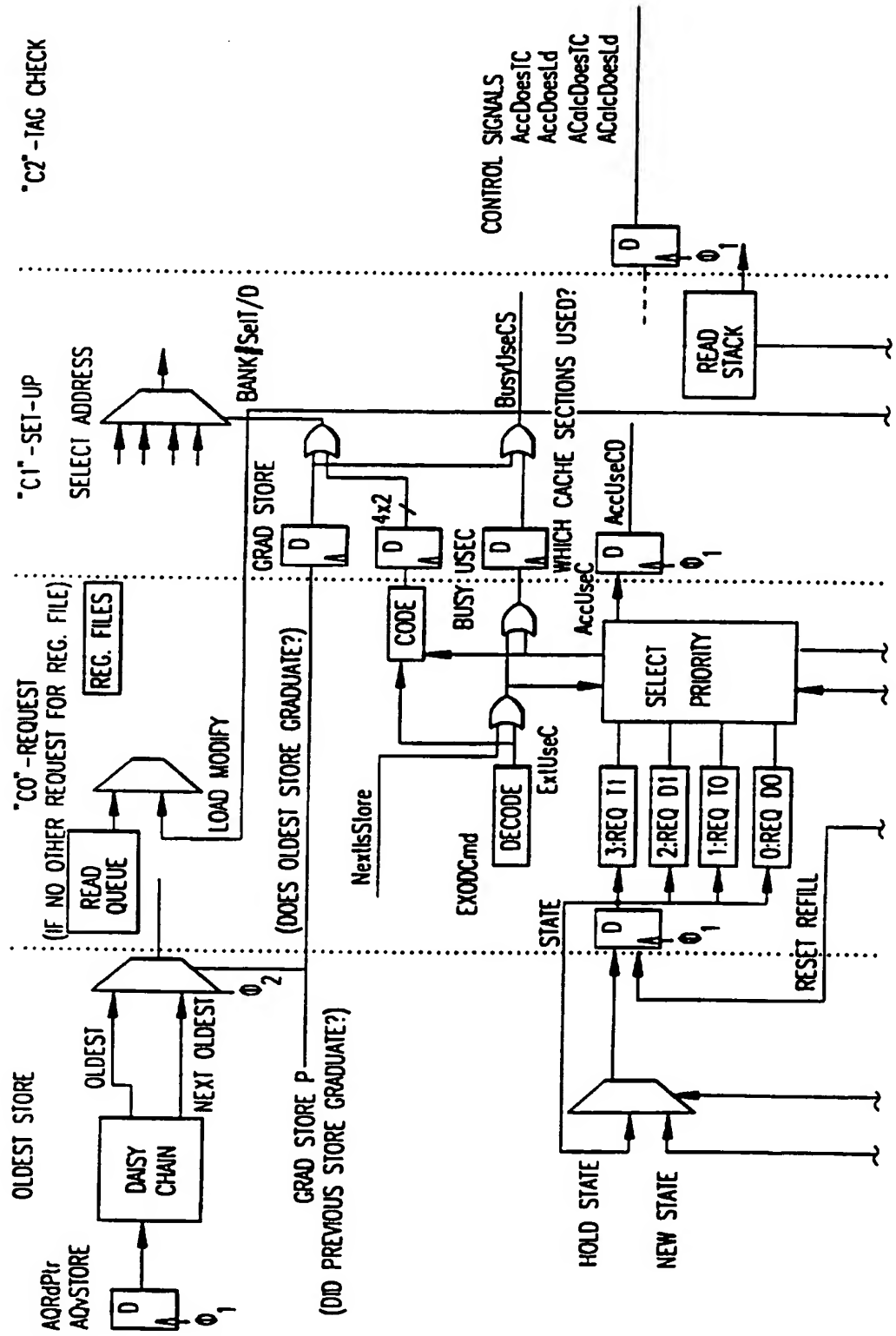

"C2"-TAG CHECK
"C1"-SET-UP
"C0"-REQUEST
(IF NO OTHER REQUEST FOR REG. FILE)
OLDEST STORE
AQRdPtr
AQvSTORE
OLDEST
NEXT OLDEST
DAISY CHAIN
READ QUEUE
REG. FILES
LOAD MODIFY
SELECT ADDRESS
BANK/SelT/D
GRAD STORE
(DOES OLDEST STORE GRADUATE?)
GRAD STORE P
(DID PREVIOUS STORE GRADUATE?)
CODE
4x2
BUSY USEC
BusyUseCS
WHICH CACHE SECTIONS USED?
AccUseCD
AccUseC
NextIsStore
EXODCmd
DECODE
ExlUseC
SELECT PRIORITY
3:REQ T1
2:REQ D1
1:REQ T0
0:REQ D0
STATE
RESET REFILL
HOLD STATE
NEW STATE
CONTROL SIGNALS
AccDoesTC
AccDoesLd
ACalcDoesTC
ACalcDoesLd
READ STACK
D
ϕ_1
ϕ_2

FIG.15A

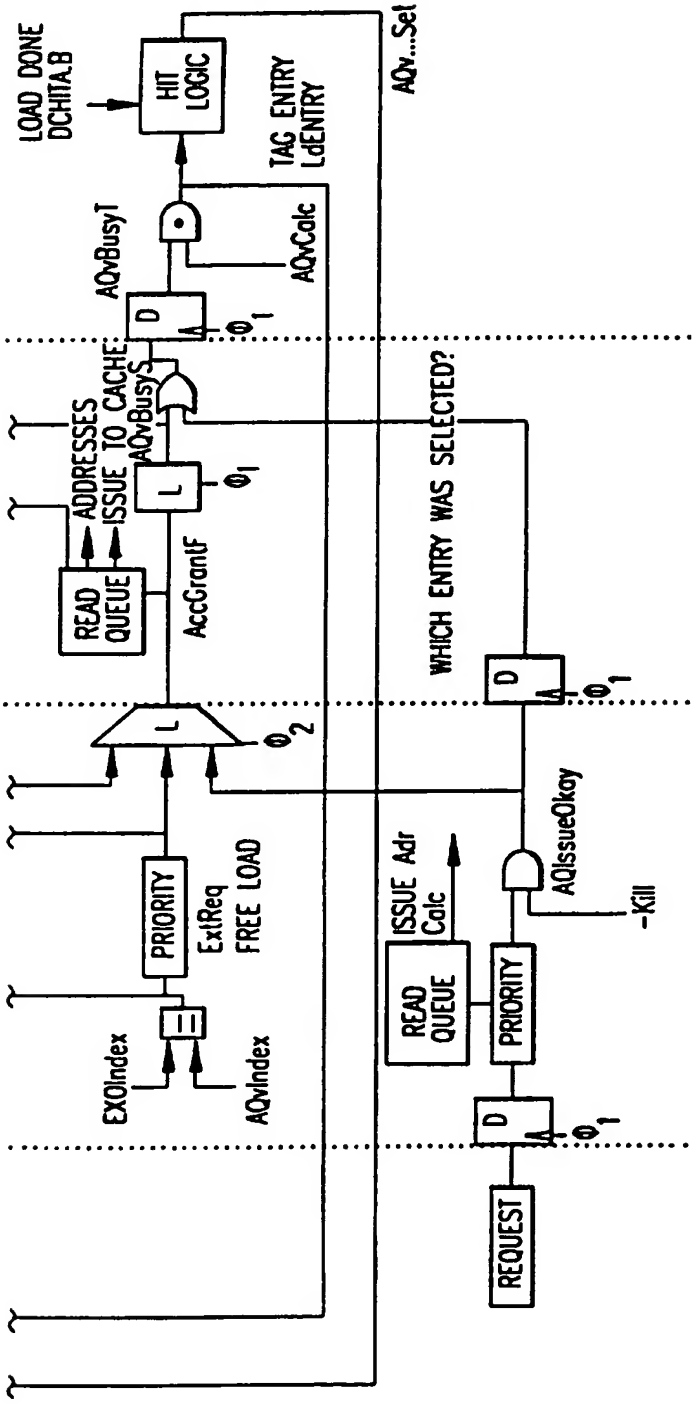

LOAD DONE
DCHITA.B
HIT LOGIC
TAG ENTRY
LdENTRY
AQv...Set
AQvBusyT
AQvCalc
D
A
Φ1
ADDRESSES
ISSUE TO CACHE
AQvBusyS
L
Φ1
READ QUEUE
AccGrantF
WHICH ENTRY WAS SELECTED?
D
A
Φ1
L
Φ2
AQIssueOkay
-Kill
ISSUE Adr
Calc
PRIORITY
ExtReq
FREE LOAD
READ QUEUE
PRIORITY
EXQIndex
AQvIndex
D
A
Φ1
REQUEST

FIG.15B

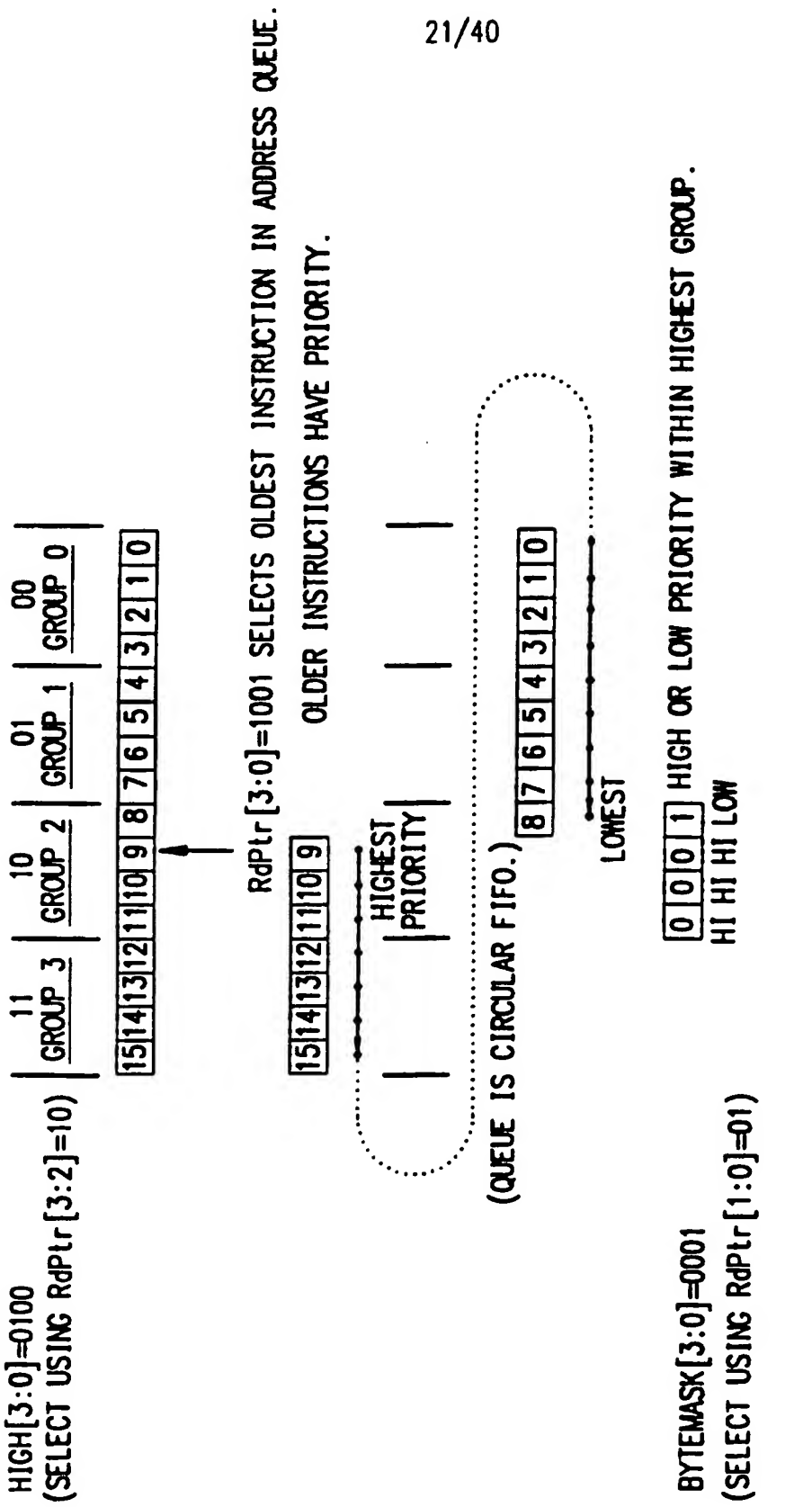

HIGH[3:0]=0100
(SELECT USING RdPtr[3:2]=10)
11 GROUP 3
10 GROUP 2
01 GROUP 1
00 GROUP 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RdPtr[3:0]=1001 SELECTS OLDEST INSTRUCTION IN ADDRESS QUEUE.
OLDER INSTRUCTIONS HAVE PRIORITY.
15 14 13 12 11 10 9
HIGHEST PRIORITY
(QUEUE IS CIRCULAR FIFO.)
8 7 6 5 4 3 2 1 0
LOWEST
BYTEMASK[3:0]=0001
(SELECT USING RdPtr[1:0]=01)
0 0 0 1
HI HI HI LOW
HIGH OR LOW PRIORITY WITHIN HIGHEST GROUP.

FIG.16

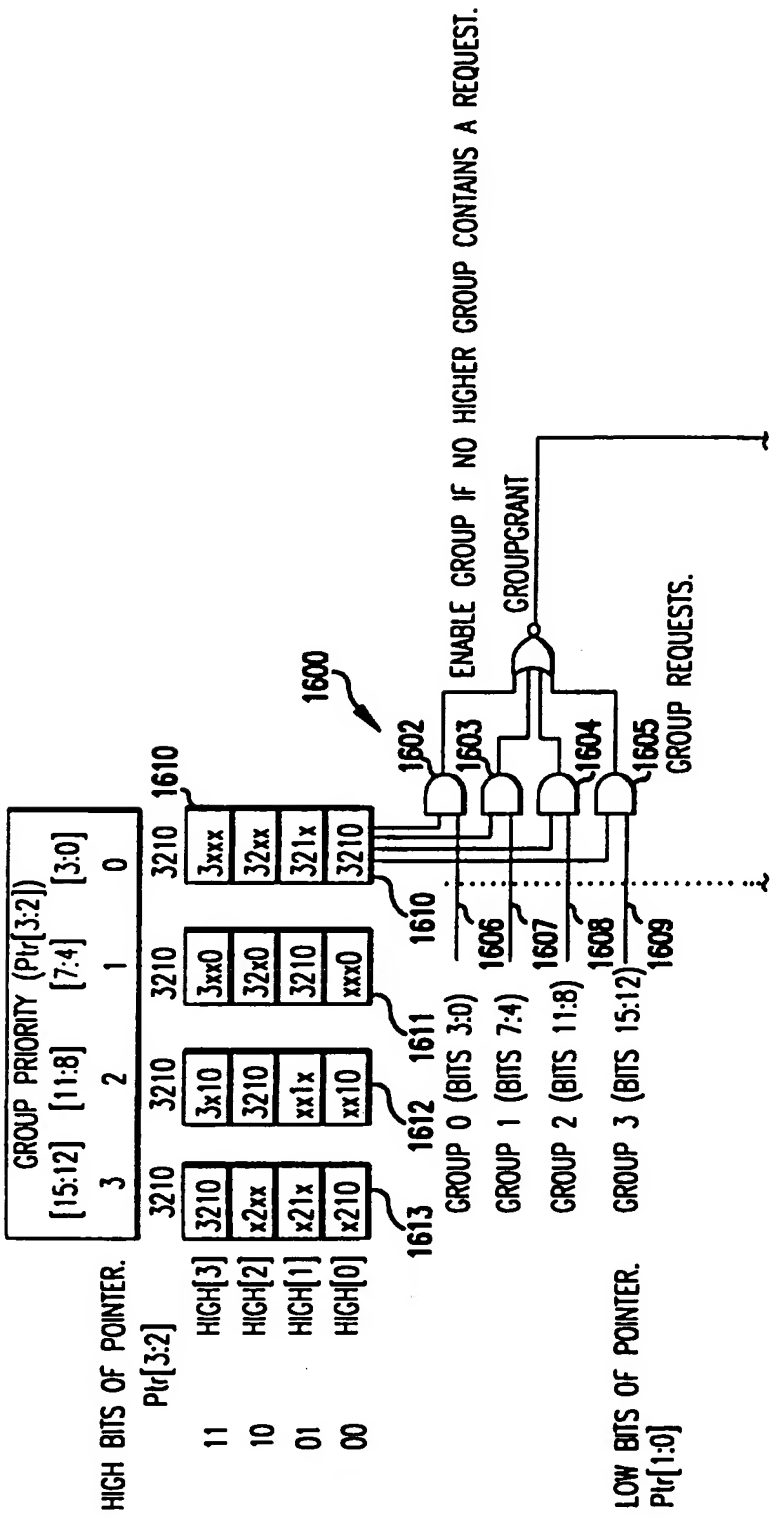

GROUP PRIORITY (Ptr[3:2])
[15:12] [11:8] [7:4] [3:0]
3 2 1 0
HIGH BITS OF POINTER. Ptr[3:2]
11
10
01
00
HIGH[3]
HIGH[2]
HIGH[1]
HIGH[0]
3210
x2xx
x21x
x210
3x10
3210
xx1x
xx10
3xx0
32x0
3210
xxx0
3xxx
32xx
321x
3210
1600
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
ENABLE GROUP IF NO HIGHER GROUP CONTAINS A REQUEST.
GROUPGRANT
GROUP REQUESTS.
GROUP 0 (BITS 3:0)
GROUP 1 (BITS 7:4)
GROUP 2 (BITS 11:8)
GROUP 3 (BITS 15:12)
LOW BITS OF POINTER. Ptr[1:0]

FIG.17A

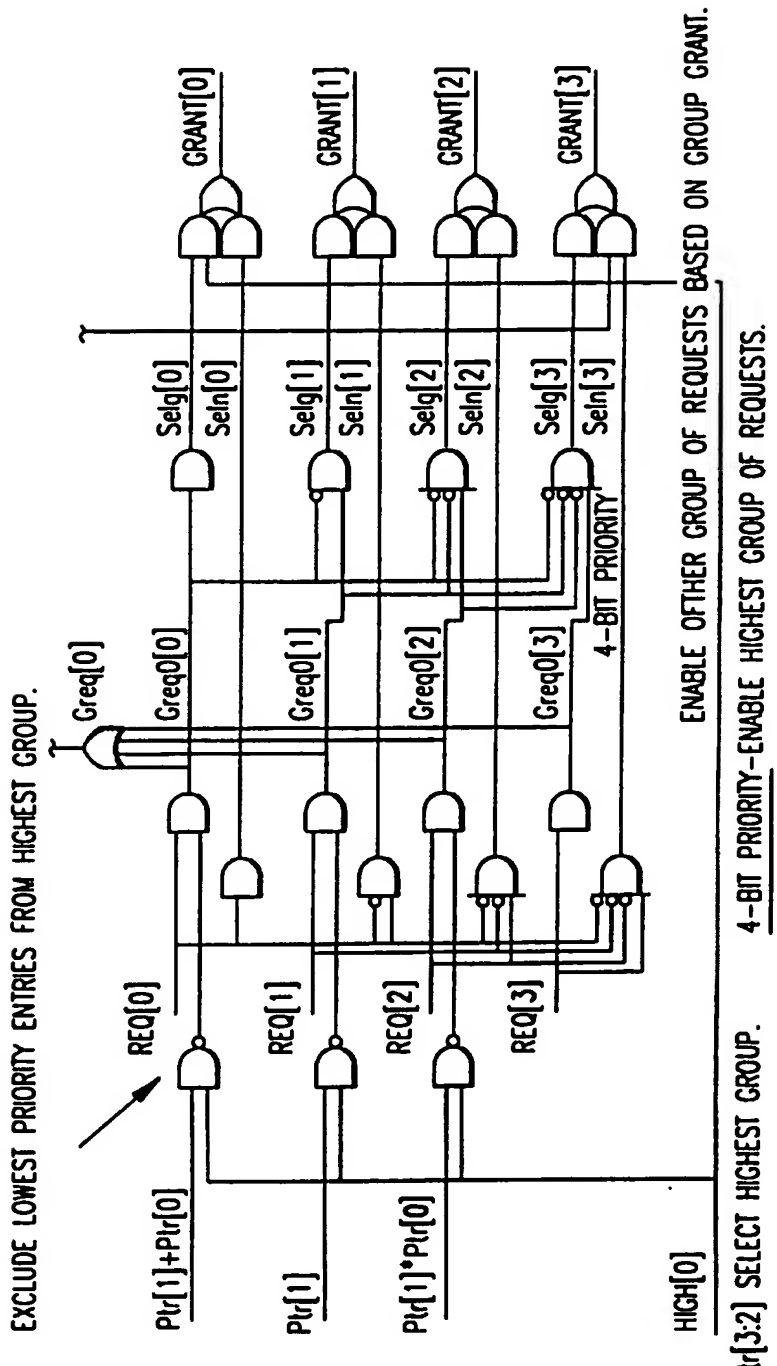

EXCLUDE LOWEST PRIORITY ENTRIES FROM HIGHEST GROUP.
Greq[0]
Greq0[0]
Greq0[1]
Greq0[2]
Greq0[3]
4-BIT PRIORITY
ENABLE OFTHER GROUP OF REQUESTS BASED ON GROUP GRANT.
GRANT[0]
GRANT[1]
GRANT[2]
GRANT[3]
Selg[0]
Seln[0]
Selg[1]
Seln[1]
Selg[2]
Seln[2]
Selg[3]
Seln[3]
REQ[0]
REQ[1]
REQ[2]
REQ[3]
Ptr[1]+Ptr[0]
Ptr[1]
Ptr[1]*Ptr[0]
HIGH[0]
Ptr[3:2] SELECT HIGHEST GROUP.
4-BIT PRIORITY-ENABLE HIGHEST GROUP OF REQUESTS.

FIG.17B

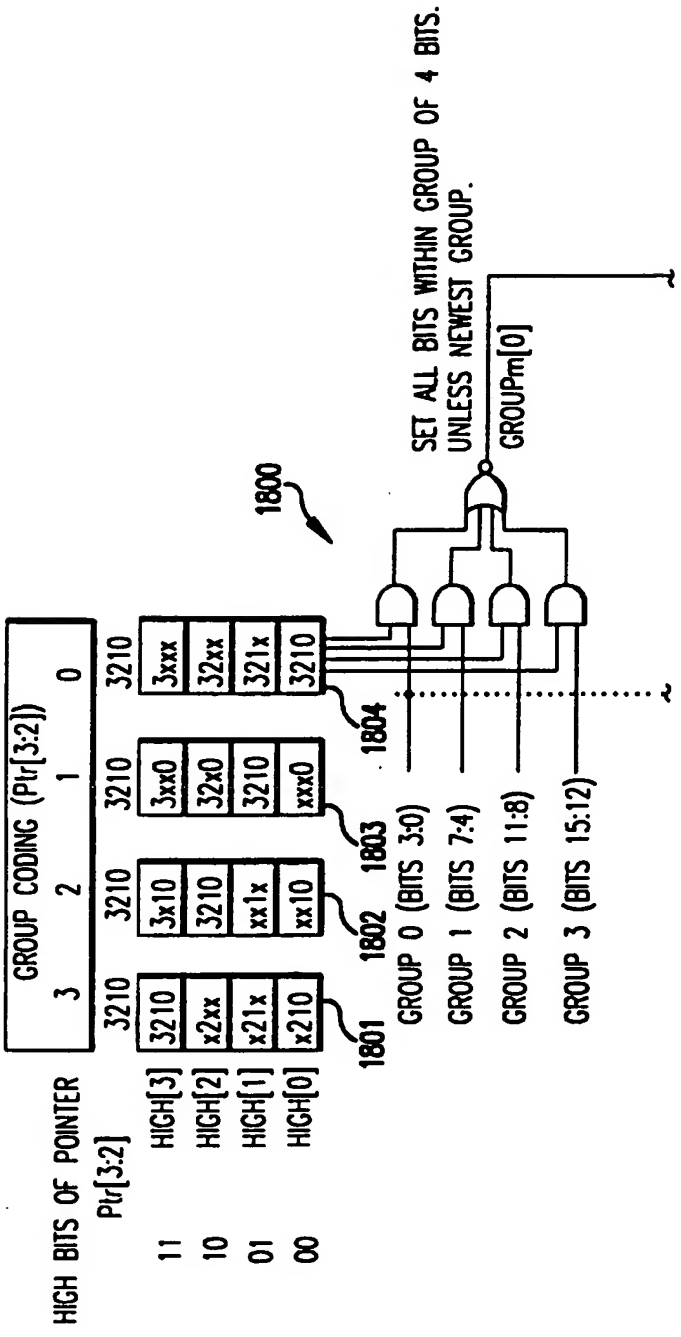

HIGH BITS OF POINTER Ptr[3:2]
11
10
01
00
HIGH[3]
HIGH[2]
HIGH[1]
HIGH[0]
GROUP CODING (Ptr[3:2])
3
2
1
0
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
3210
1801
1802
1803
1804
1800
GROUP 0 (BITS 3:0)
GROUP 1 (BITS 7:4)
GROUP 2 (BITS 11:8)
GROUP 3 (BITS 15:12)
SET ALL BITS WITHIN GROUP OF 4 BITS.
UNLESS NEWEST GROUP.
GROUPm[0]

FIG.18A

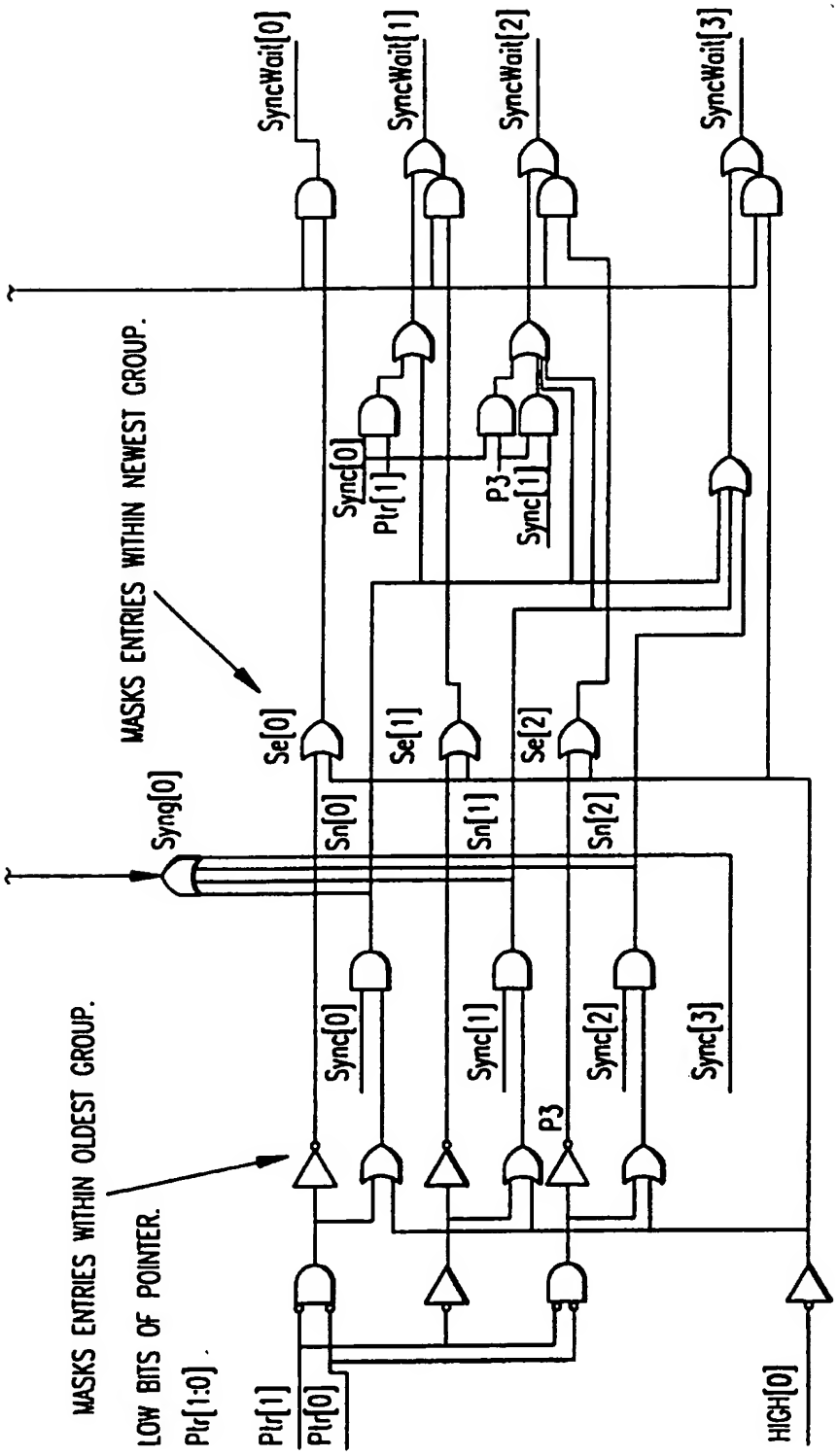

SyncWait[0]
SyncWait[1]
SyncWait[2]
SyncWait[3]
MASKS ENTRIES WITHIN NEWEST GROUP.
Sync[0]
Ptr[1]
P3
Sync[1]
Se[0]
Se[1]
Se[2]
Syng[0]
Sn[0]
Sn[1]
Sn[2]
Sync[0]
Sync[1]
Sync[2]
Sync[3]
P3
MASKS ENTRIES WITHIN OLDEST GROUP.
LOW BITS OF POINTER.
Ptr[1:0]
Ptr[1]
Ptr[0]
HIGH[0]

FIG.18B

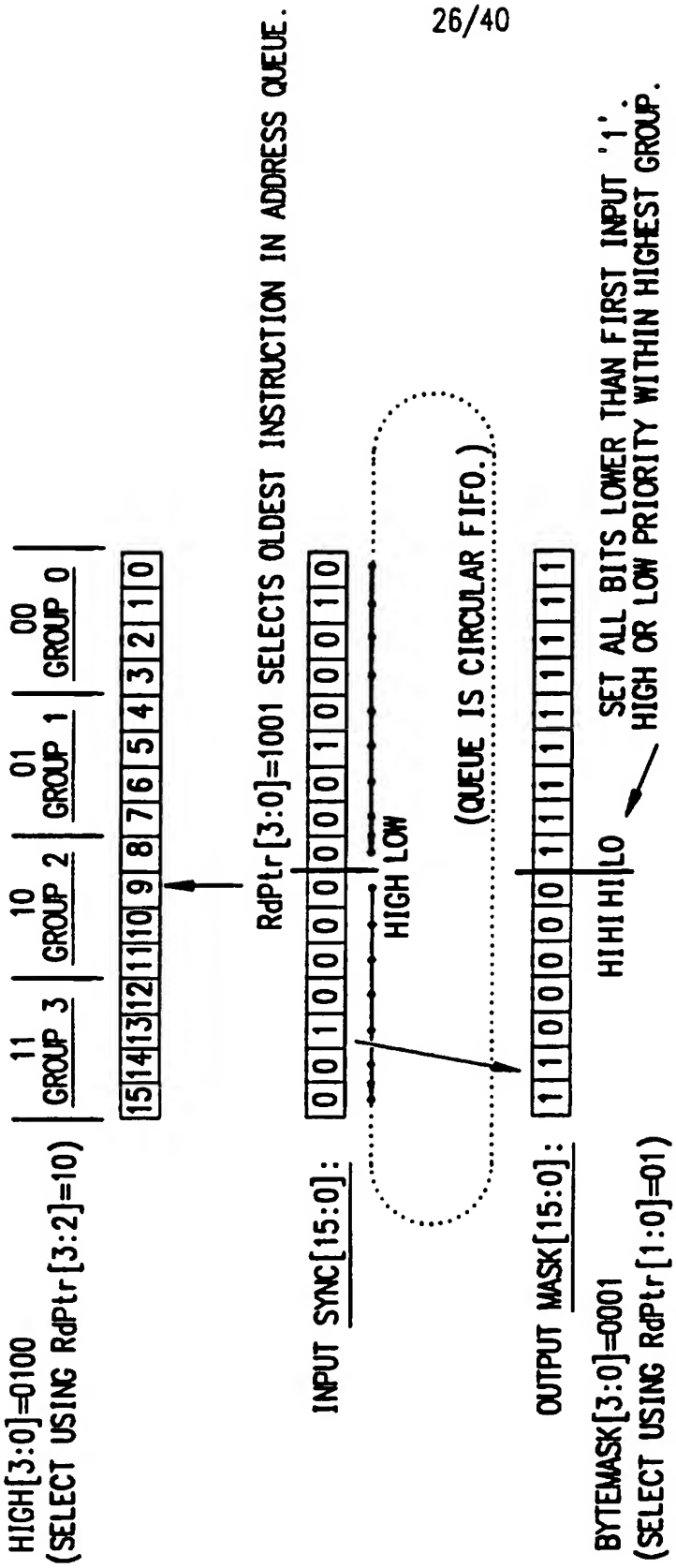

HIGH[3:0]=0100
(SELECT USING RdPtr[3:2]=10)
11 GROUP 3
10 GROUP 2
01 GROUP 1
00 GROUP 0
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
RdPtr[3:0]=1001 SELECTS OLDEST INSTRUCTION IN ADDRESS QUEUE.
INPUT SYNC[15:0]: 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0
HIGH LOW
(QUEUE IS CIRCULAR FIFO.)
OUTPUT MASK[15:0]: 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1
HI HI HI LO
SET ALL BITS LOWER THAN FIRST INPUT '1'.
HIGH OR LOW PRIORITY WITHIN HIGHEST GROUP.
BYTEMASK[3:0]=0001
(SELECT USING RdPtr[1:0]=01)

FIG.19

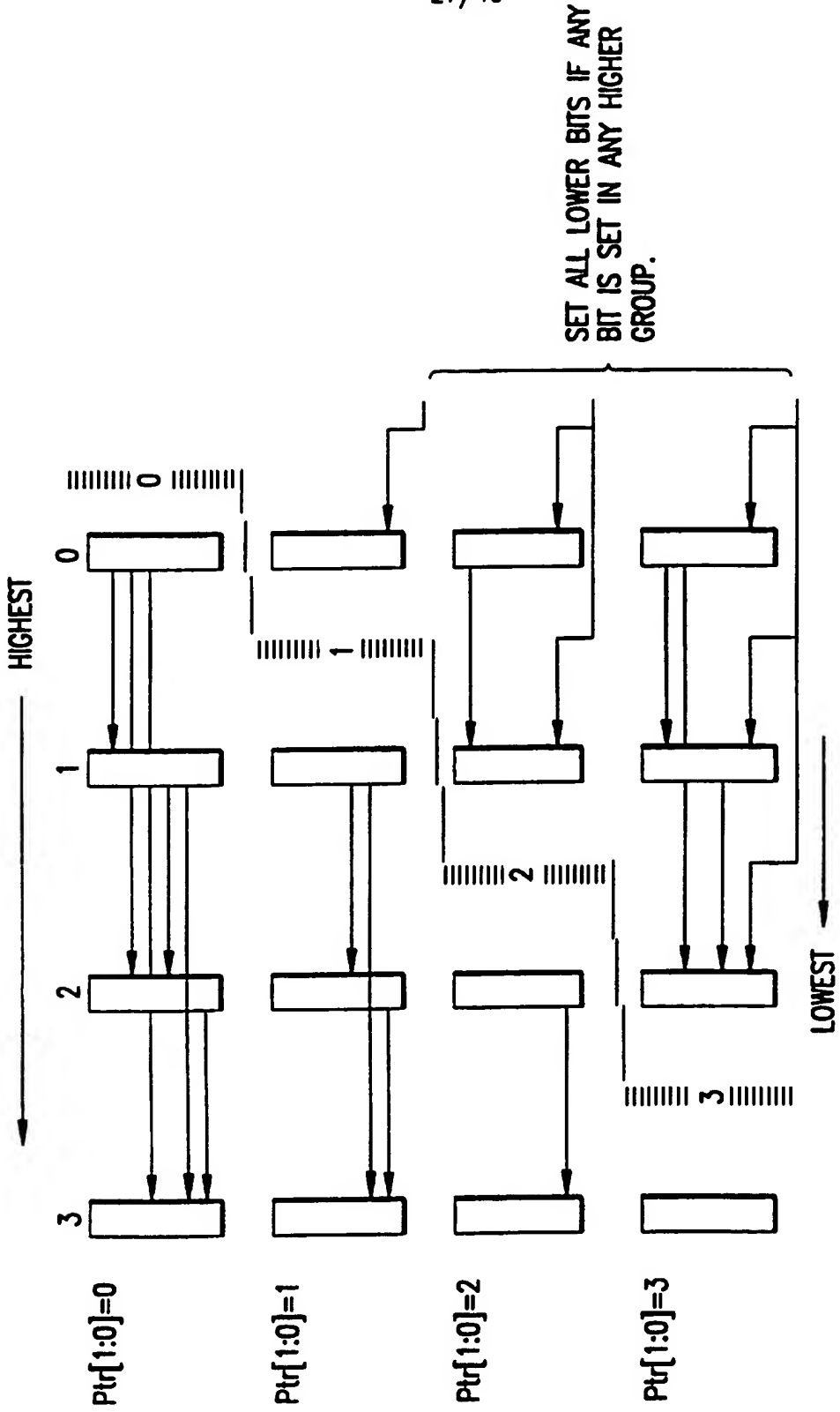

SET ALL LOWER BITS IF ANY BIT IS SET IN ANY HIGHER GROUP.
HIGHEST
LOWEST
0
1
2
3
Ptr[1:0]=0
Ptr[1:0]=1
Ptr[1:0]=2
Ptr[1:0]=3

FIG.20

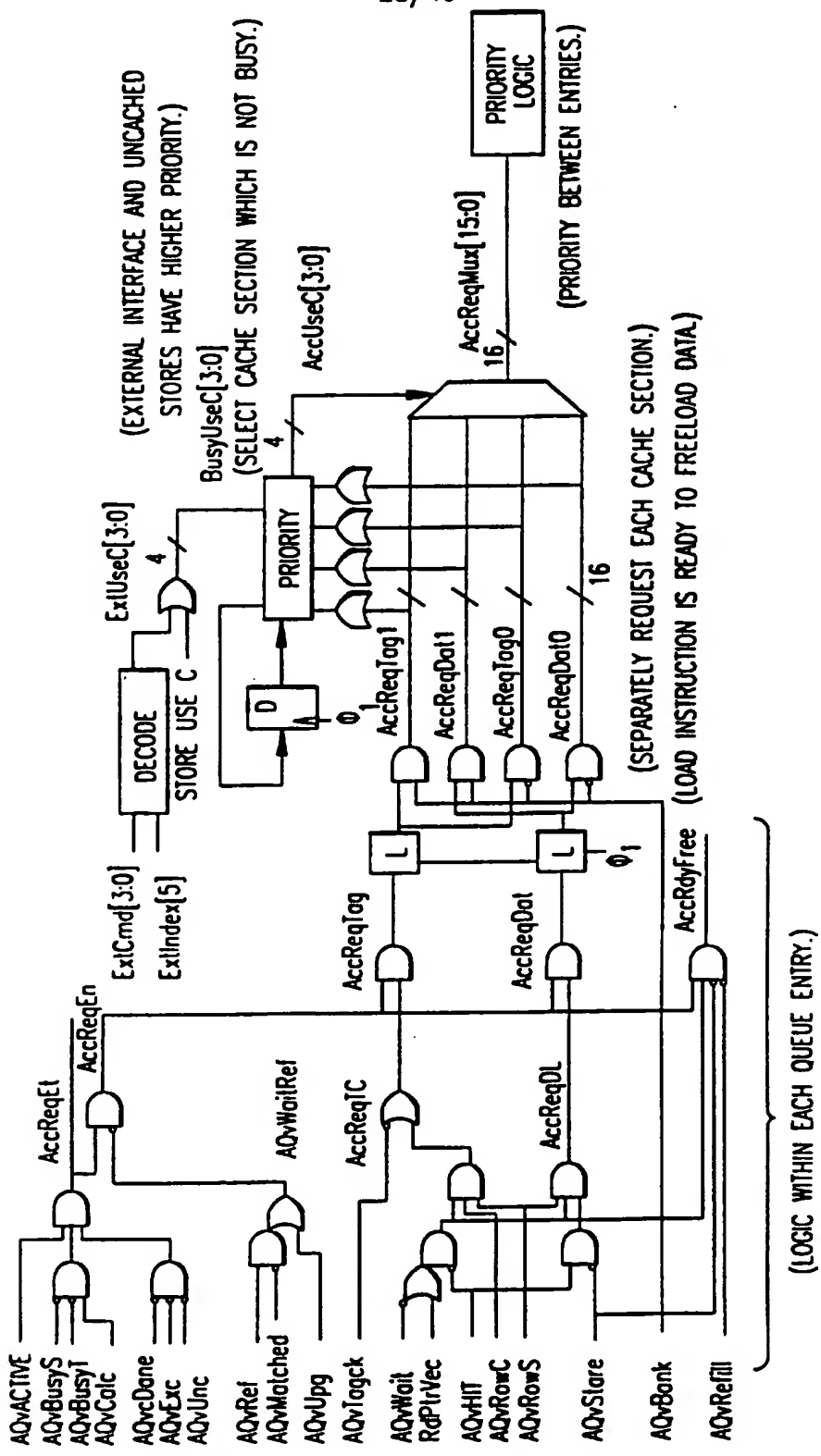
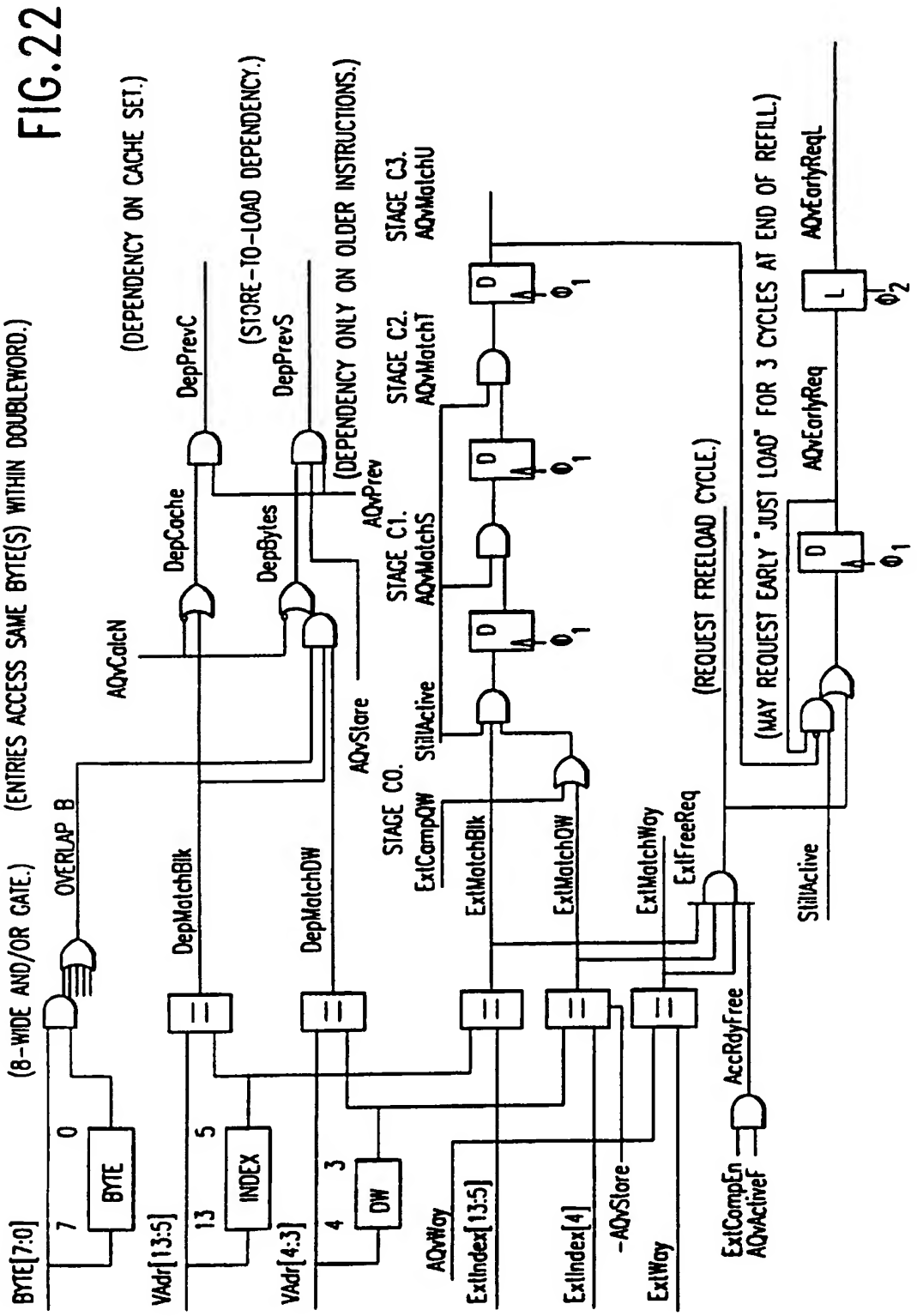

AQvACTIVE
AQvBusyS
AQvBusyT
AQvCalc
AQvcDone
AQvExc
AQvUnc
AQvRef
AQvMatched
AQvUpg
AQvTagck
AQvWait
RdPtrVec
AQvHIT
AQvRowC
AQvRowS
AQvSlare
AQvBank
AQvRefill
AccReqEt
AccReqEn
AQvWaitRef
AccReqIC
AccReqDL
ExtCmd[3:0]
ExtIndex[5]
AccReqTag
AccReqDat
AccRdyFree
(LOGIC WITHIN EACH QUEUE ENTRY.)
DECODE
STORE USE C
ExtUseC[3:0]
BusyUseC[3:0]
(EXTERNAL INTERFACE AND UNCACHED
STORES HAVE HIGHER PRIORITY.)
(SELECT CACHE SECTION WHICH IS NOT BUSY.)
PRIORITY
AccUseC[3:0]
AccReqTag1
AccReqDat1
AccReqTag0
AccReqDat0
AccReqMux[15:0]
PRIORITY LOGIC
(PRIORITY BETWEEN ENTRIES.)
(SEPARATELY REQUEST EACH CACHE SECTION.)
(LOAD INSTRUCTION IS READY TO FREELOAD DATA.)

FIG.21

FIG.22


(8-WIDE AND/OR GATE.)
(ENTRIES ACCESS SAME BYTE(S) WITHIN DOUBLEWORD.)
(DEPENDENCY ON CACHE SET.)
(STORE-TO-LOAD DEPENDENCY.)
(DEPENDENCY ONLY ON OLDER INSTRUCTIONS.)
(REQUEST FREELOAD CYCLE.)
(MAY REQUEST EARLY "JUST LOAD" FOR 3 CYCLES AT END OF REFILL.)
BYTE[7:0]
OVERLAP B
BYTE
INDEX
DW
VAdr[13:5]
VAdr[4:3]
AQvCalcN
DepCache
DepBytes
DepPrevC
DepPrevS
AQvPrev
AQvStore
DepMatchBlk
DepMatchDW
STAGE C0.
STAGE C1.
STAGE C2.
STAGE C3.
StillActive
ExtCompDW
ExtMatchBlk
ExtMatchDW
ExtMatchWay
ExtFreeReq
AQvMatchS
AQvMatchT
AQvMatchU
AQvEarlyReq
AQvEarlyReqL
AQvWay
ExtIndex[13:5]
ExtIndex[4]
-AQvStore
ExtWay
AccRdyFree
ExtCompEn
AQvActiveF

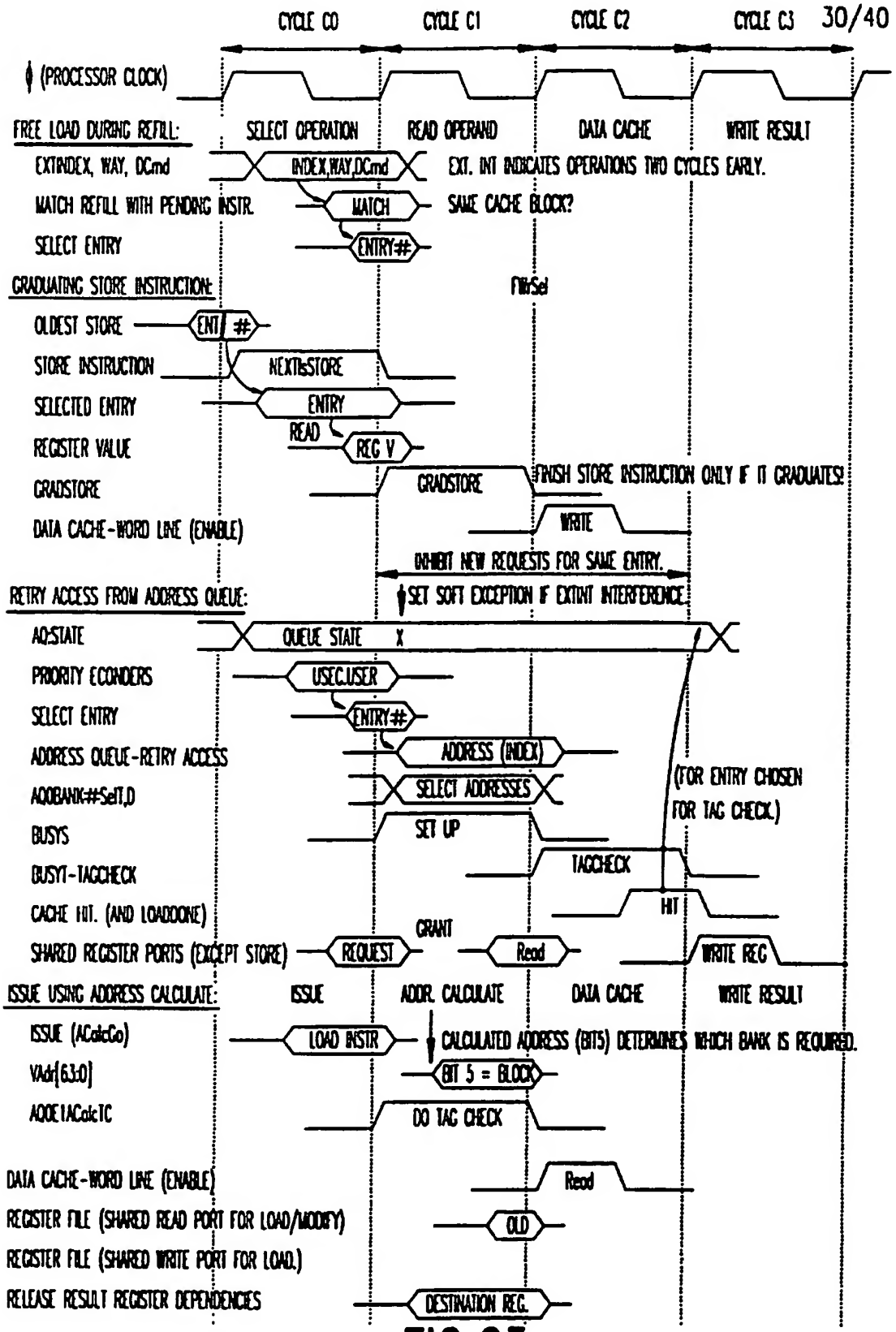

CYCLE C0
CYCLE C1
CYCLE C2
CYCLE C3
30/40
(PROCESSOR CLOCK)
FREE LOAD DURING REFILL:
SELECT OPERATION
READ OPERAND
DATA CACHE
WRITE RESULT
EXTINDEX, WAY, DCmd
INDEX,WAY,DCmd
EXT. INT INDICATES OPERATIONS TWO CYCLES EARLY.
MATCH REFILL WITH PENDING INSTR.
MATCH
SAME CACHE BLOCK?
SELECT ENTRY
ENTRY#
GRADUATING STORE INSTRUCTION:
FWrSel
OLDEST STORE
ENT #
STORE INSTRUCTION
NEXTisSTORE
SELECTED ENTRY
ENTRY
READ
REGISTER VALUE
REG V
GRADSTORE
GRADSTORE
FINISH STORE INSTRUCTION ONLY IF IT GRADUATES!
DATA CACHE-WORD LINE (ENABLE)
WRITE
INHIBIT NEW REQUESTS FOR SAME ENTRY.
RETRY ACCESS FROM ADDRESS QUEUE:
SET SOFT EXCEPTION IF EXTINT INTERFERENCE
AQ:STATE
QUEUE STATE X
PRIORITY ECONDERS
USEC,USER
SELECT ENTRY
ENTRY#
ADDRESS QUEUE-RETRY ACCESS
ADDRESS (INDEX)
AQDBANK#SelT,D
SELECT ADDRESSES
(FOR ENTRY CHOSEN FOR TAG CHECK.)
BUSYS
SET UP
BUSYT-TAGCHECK
TAGCHECK
CACHE HIT. (AND LOADDONE)
HIT
SHARED REGISTER PORTS (EXCEPT STORE)
GRANT
REQUEST
Read
WRITE REG
ISSUE USING ADDRESS CALCULATE:
ISSUE
ADDR. CALCULATE
DATA CACHE
WRITE RESULT
ISSUE (ACalcCo)
LOAD INSTR
CALCULATED ADDRESS (BIT5) DETERMINES WHICH BANK IS REQUIRED.
VAdr[63:0]
BIT 5 = BLOCK
ADDETACalcTC
DO TAG CHECK
DATA CACHE-WORD LINE (ENABLE)
Read
REGISTER FILE (SHARED READ PORT FOR LOAD/MODIFY)
OLD
REGISTER FILE (SHARED WRITE PORT FOR LOAD)
RELEASE RESULT REGISTER DEPENDENCIES
DESTINATION REG.

FIG.23

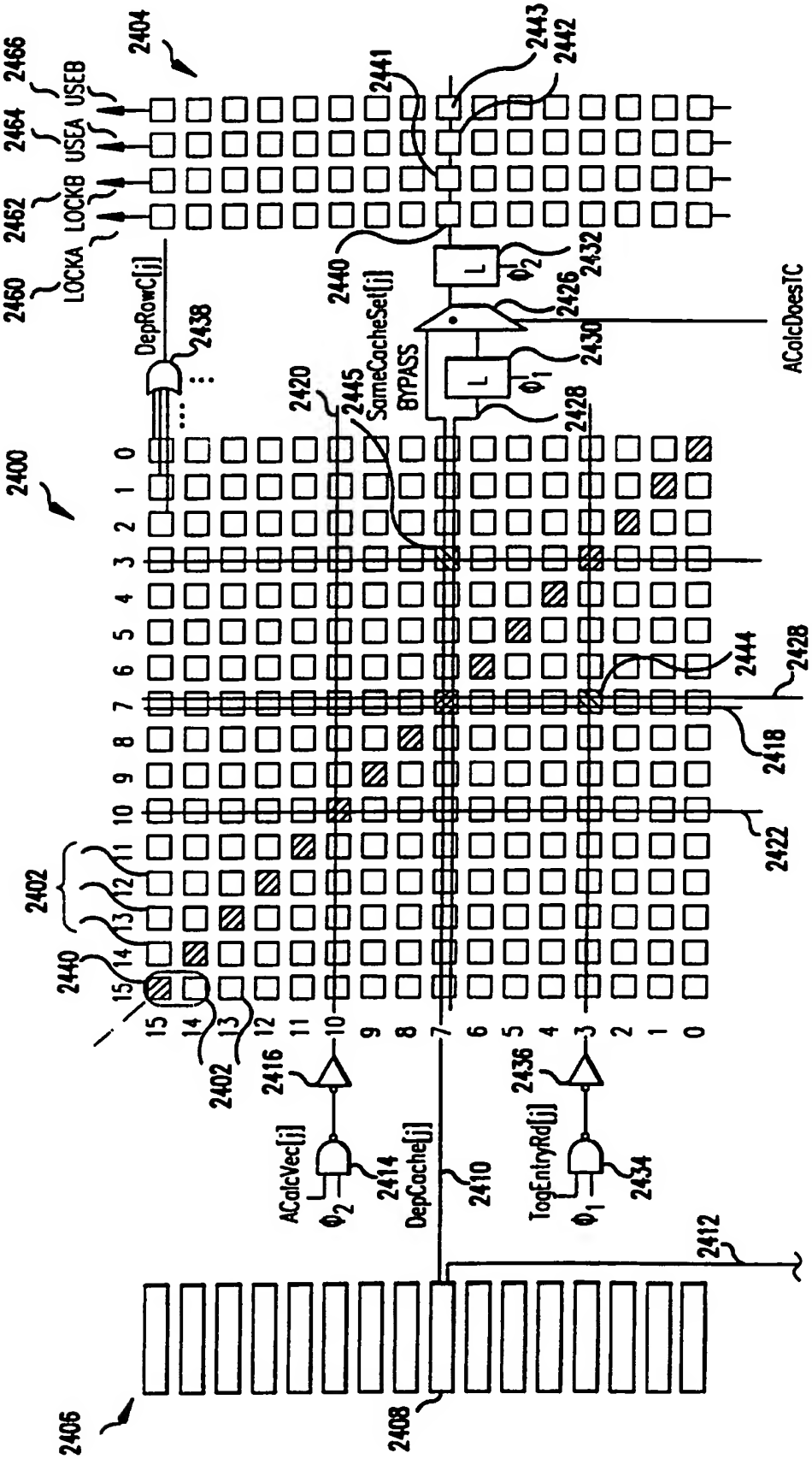
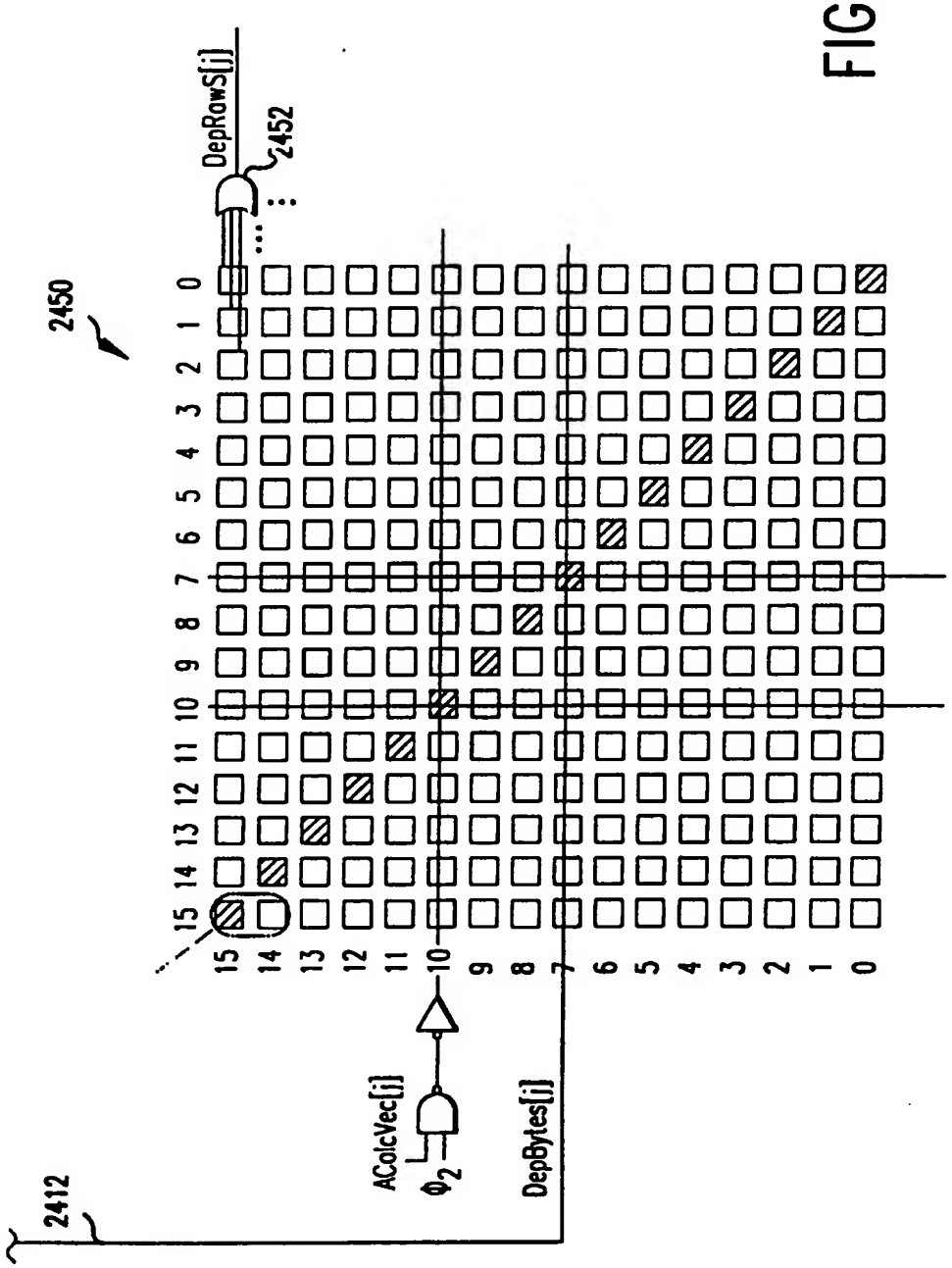

2404
2466
USEB
2464
USEA
2462
LOCKB
2460
LOCKA
2441
2443
2442
2440
2432
2426
2430
2428
2420
2445
SameCacheSet[j]
BYPASS
DepRow[j]
2438
ACalcDoesTC
2400
2444
2428
2418
2422
2402
2416
2436
ACalcVec[j]
2414
DepCache[j]
2410
TagEntryRd[j]
2434
2412
2406
2408

FIG.24A

FIG.24B


DepRows[j]
2452
2450
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
ACalcVec[j]
Φ_2
DepBytes[j]
2412

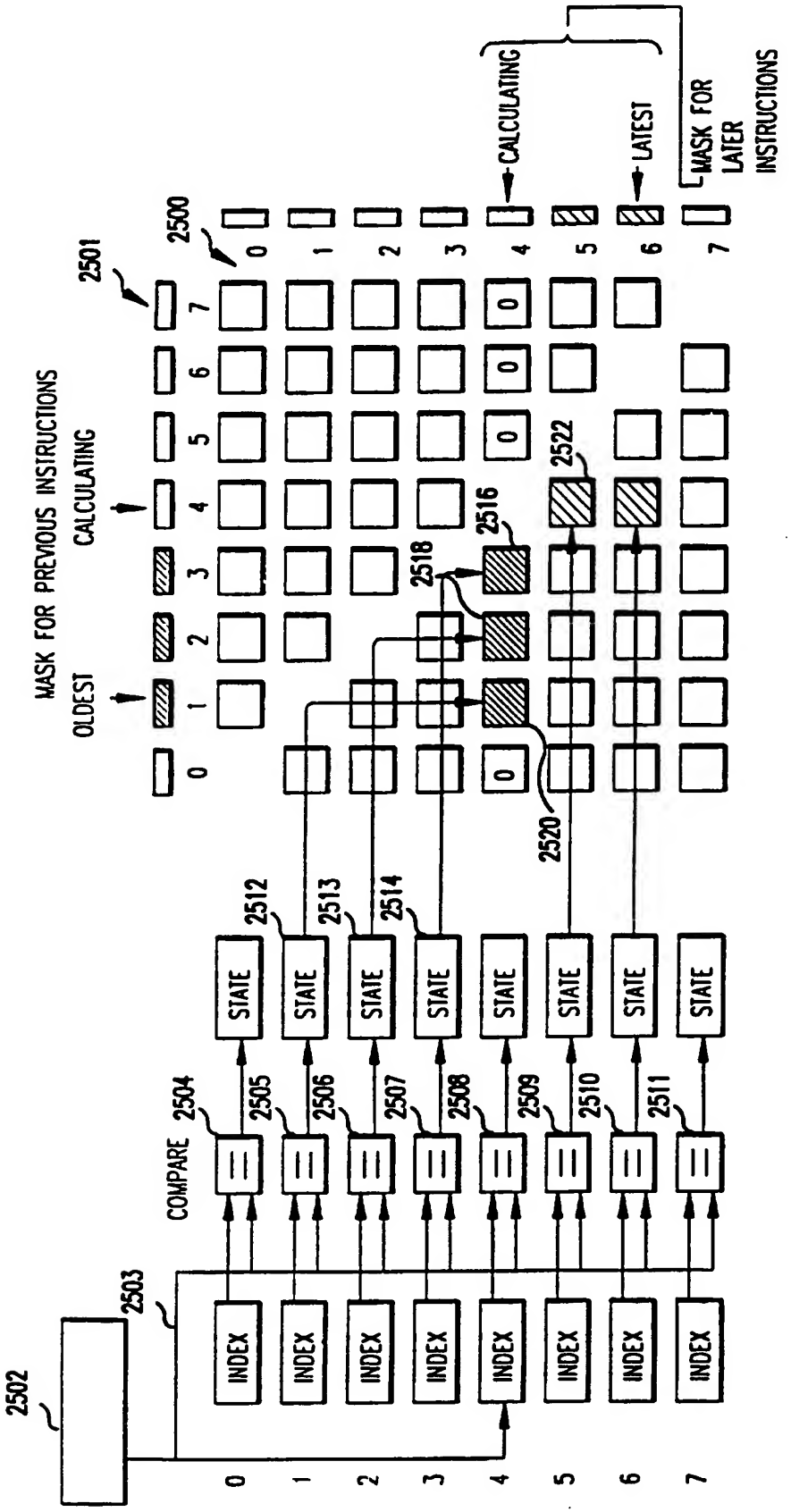

MASK FOR PREVIOUS INSTRUCTIONS
OLDEST
CALCULATING
2500
2501
CALCULATING
LATEST
MASK FOR LATER INSTRUCTIONS
2516
2518
2522
2520
2512
2513
2514
STATE
COMPARE
2504
2505
2506
2507
2508
2509
2510
2511
INDEX
2503
2502
0 1 2 3 4 5 6 7

FIG.25

FIG.26

DEPENDENCY CHECKS DURING TAG CHECK CYCLES																			
		ARRAY STATUS				CACHE TAGS				ACTION				CACHE	
LOCK		USE		DEP		HIT	LRU	STATE		LOCK		USE		ACTION	COMMENTS
A	B	A	B	D	L	A	B	•	A	B	A	B	A	B	
															FIRST ACCESS TO THIS SET.
0	0	0	0	--	L	1	0	--	E	--	C	--	--	--	HIT. LOCK INTO SIDE A, UNLESS STORED DEPENDENCY.
0	0	0	0	--	L	0	1	--	--	E	--	C	--	--	HIT. LOCK INTO SIDE B, UNLESS STORE DEPENDENCY.
0	0	0	0	--	0	0	0	0	A	--	S	--	--	--	BEGIN REFILL OF CACHE INTO SIDE A. (LRU=0.)
0	0	0	0	--	0	0	0	--	A	R	S	--	--	--	REFILL (IGNORE LRU BIT IF B IS NOT AVAILABLE.)
0	0	0	0	--	0	0	0	1	--	A	--	S	--	--	BEGIN REFILL OF CACHE INTO SIDE B. (LRU=0.)
0	0	0	0	--	0	0	0	--	R	A	--	S	--	--	REFILL (IGNORE LRU BIT IF A IS NOT AVAILABLE.)
0	0	0	0	--	--	0	0	--	R	R	--	--	--	--	WAIT BOTH CACHE BLOCKS ARE BUSY BUT NOT LOCKED.
0	0	0	0	--	1	0	0	0	A	A	--	--	--	--	REFILL BEGIN REFILL OF CACHE INTO SIDE A. (LRU=0.)
0	0	0	0	--	1	0	0	1	A	A	--	--	--	--	REFILL BEGIN REFILL OF CACHE INTO SIDE B. (LRU=1.)
0	0	0	0	--	1	0	0	0	R	--	--	--	--	--	WAIT WAIT IF EITHER SIDE IS BEING REFILLED.
0	0	0	0	--	1	0	0	--	R	--	--	--	--	--	WAIT
															SECOND ACCESS TO THIS SET (SIDE A).
1	0	0	--	0	--	1	0	--	E	--	S	--	--	--	-- (FIRST ACCESS IS HIT ON SIDE A. DUPLICATE FOR SIDE B.) HIT. BLOCK IS ALREADY LOCKED. LOCK INTO SIDE A.
															CHECK FOR STORE-TO-LOAD BYPASS.
1	0	0	0	0	--	0	0	--	E	A	--	--	--	S	REFILL MISS. REFILL SEQUENTIAL USE OF SIDE B.
1	0	0	0	0	--	0	0	--	E	R	--	--	--	--	WAIT MISS. OTHER BLOCK IS BUSY REFILLING UNWANTED DATA.
1	0	0	0	0	--	0	1	--	E	E	--	--	--	S	-- HIT. SEQUENTIAL USE OF SIDE B.
1	0	0	0	1	--	0	--	--	E	--	--	--	--	--	WAIT MISS. MUST WAIT FOR DEPENDENCY TO BE REMOVED.
															MISS. SET IS BUSY. WAIT FOR PREVIOUS TO GRADUATE.
1	0	0	1	1	--	0	--	--	E	--	--	--	--	--	WAIT (DEPENDENT ON SET WHICH SET "USE" BIT.)

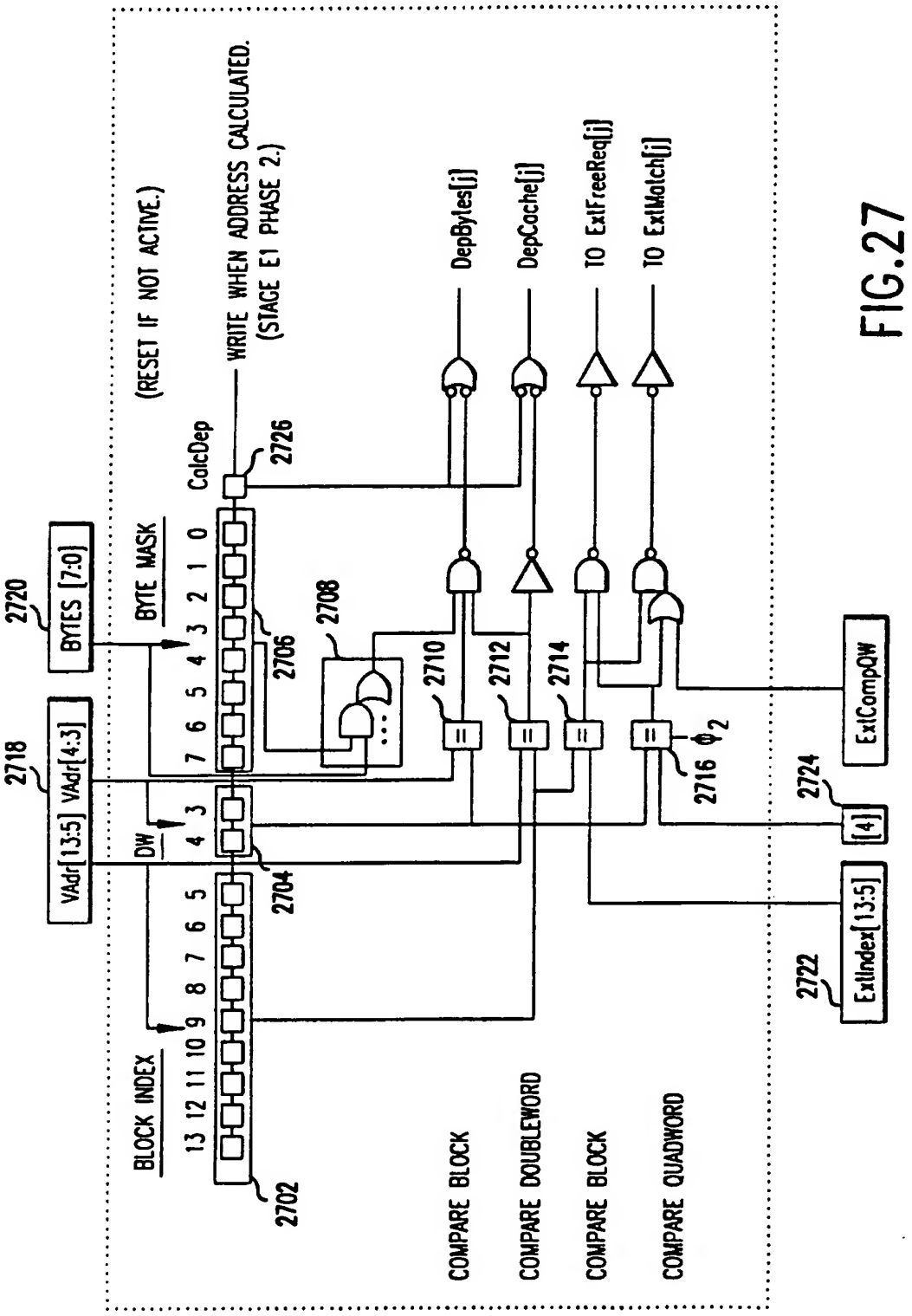

2720
BYTES [7:0]
BYTE MASK
7 6 5 4 3 2 1 0
2706
2708
2710
2712
2714
2716 ϕ_2
2718
VAdr[13:5] VAdr[4:3]
DW
4 3
2704
BLOCK INDEX
13 12 11 10 9 8 7 6 5
2702
(RESET IF NOT ACTIVE.)
WRITE WHEN ADDRESS CALCULATED.
(STAGE E1 PHASE 2.)
CalcDep
2726
DepBytes[j]
DepCache[j]
TO ExtFreeReq[j]
TO ExtWatch[j]
COMPARE BLOCK
COMPARE DOUBLEWORD
COMPARE BLOCK
COMPARE QUADWORD
2724
[4]
ExtCompQW
2722
ExtIndex[13:5]

FIG.27

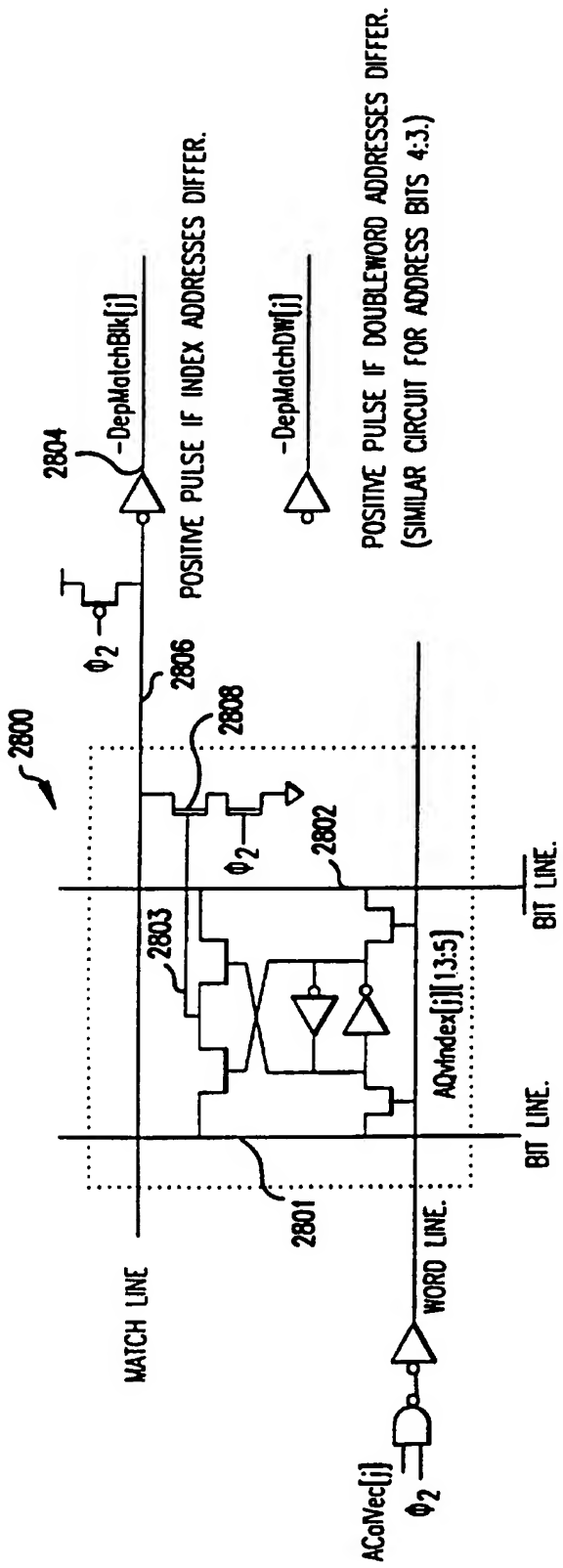

2800
2804
-DepMatchBlk[j]
POSITIVE PULSE IF INDEX ADDRESSES DIFFER.
-DepMatchDW[j]
POSITIVE PULSE IF DOUBLEWORD ADDRESSES DIFFER.
(SIMILAR CIRCUIT FOR ADDRESS BITS 4:3.)
ϕ_2
2806
2808
2802
2803
BIT LINE.
AQvIndex[j][13:5]
2801
MATCH LINE
WORD LINE.
AQoVec[j]

FIG.28

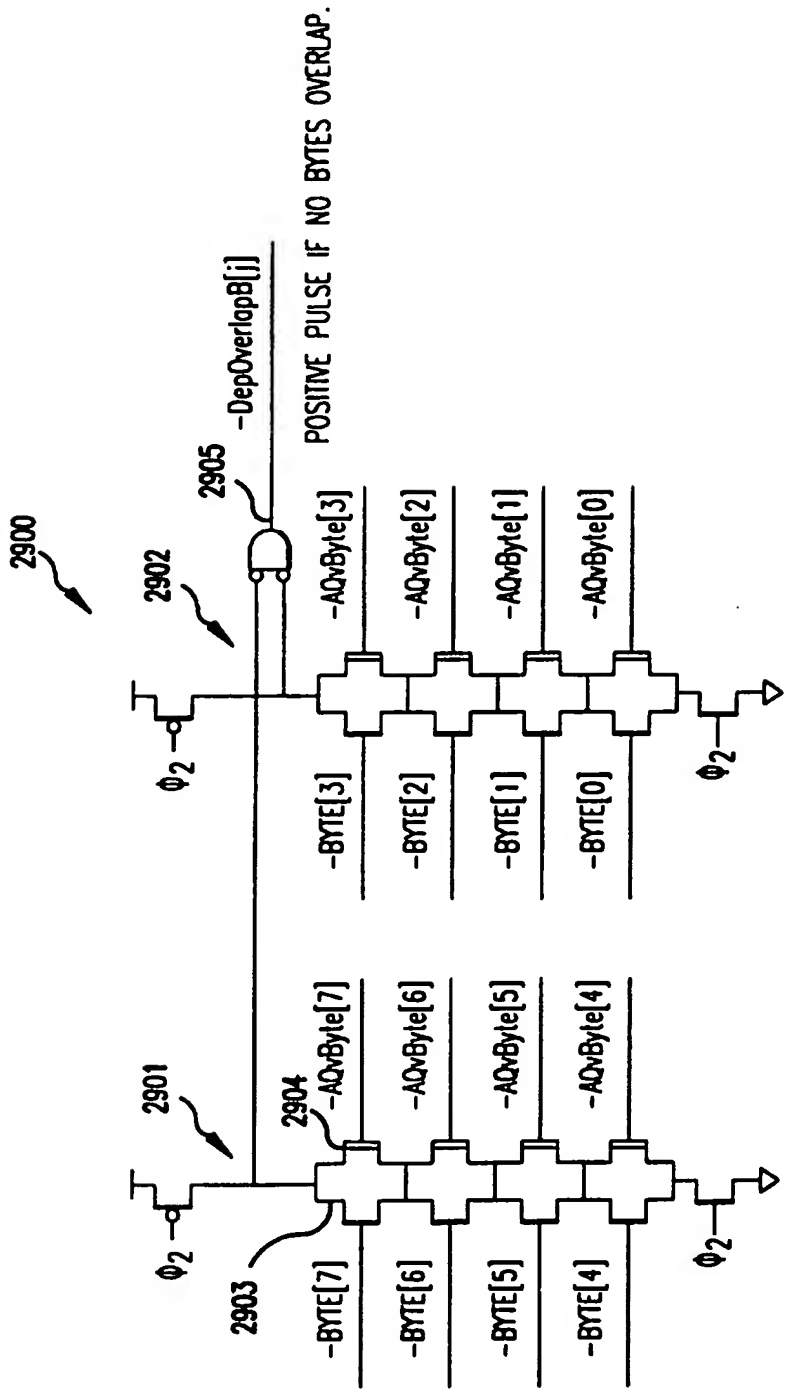

2900
2902
2905
-DepOverlapB[j]
POSITIVE PULSE IF NO BYTES OVERLAP.
-AOvByte[3]
-AOvByte[2]
-AOvByte[1]
-AOvByte[0]
ϕ_2
-BYTE[3]
-BYTE[2]
-BYTE[1]
-BYTE[0]
ϕ_2
2901
2904
-AOvByte[7]
-AOvByte[6]
-AOvByte[5]
-AOvByte[4]
ϕ_2
2903
-BYTE[7]
-BYTE[6]
-BYTE[5]
-BYTE[4]
ϕ_2

FIG.29

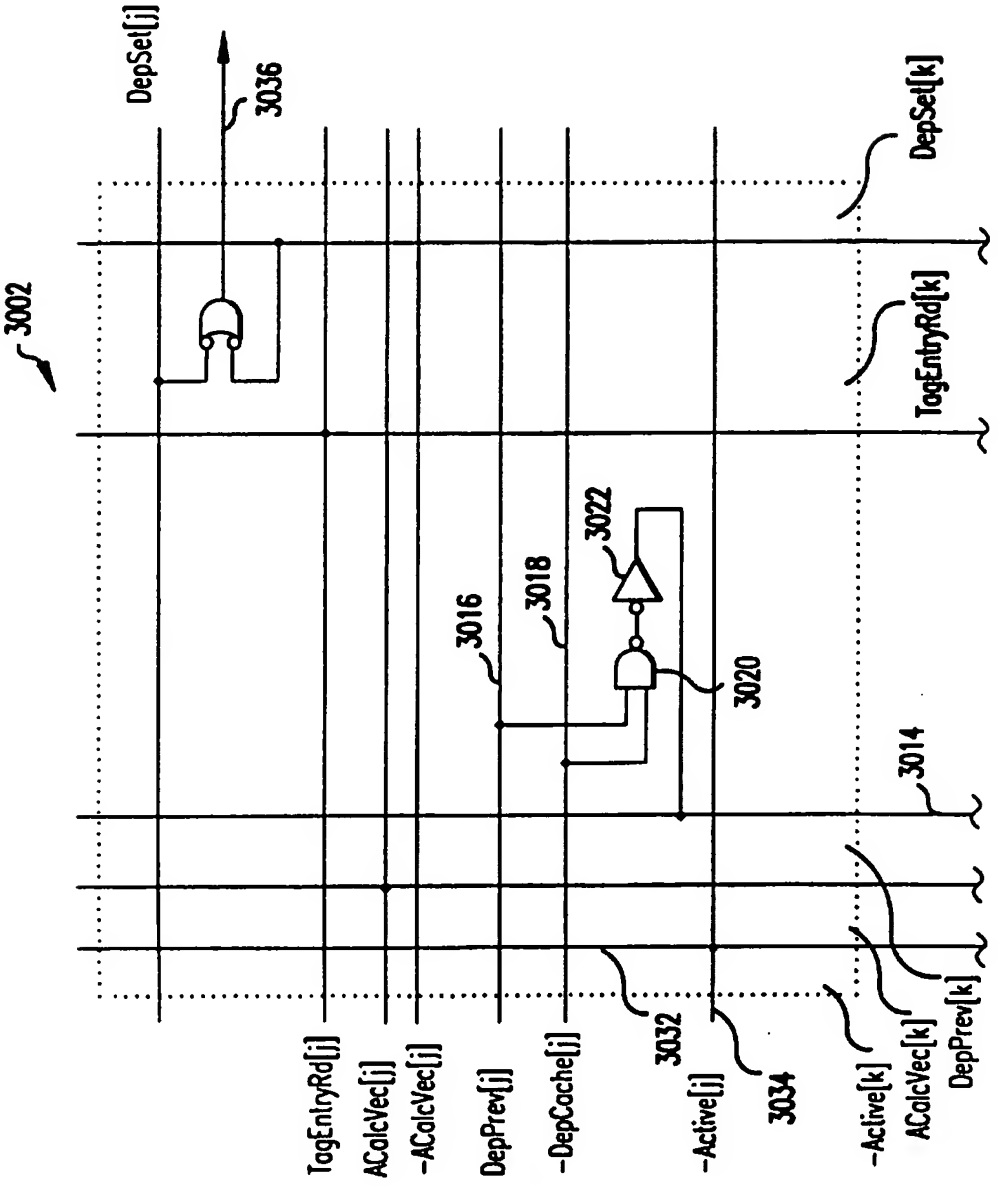

DepSet[j]
3036
3002
DepSet[k]
TagEntryRd[k]
3022
3018
3016
3020
3014
TagEntryRd[j]
ACalcVec[j]
-ACalcVec[j]
DepPrev[j]
-DepCache[j]
3032
-Active[j]
3034
-Active[k]
ACalcVec[k]
DepPrev[k]

FIG.30A

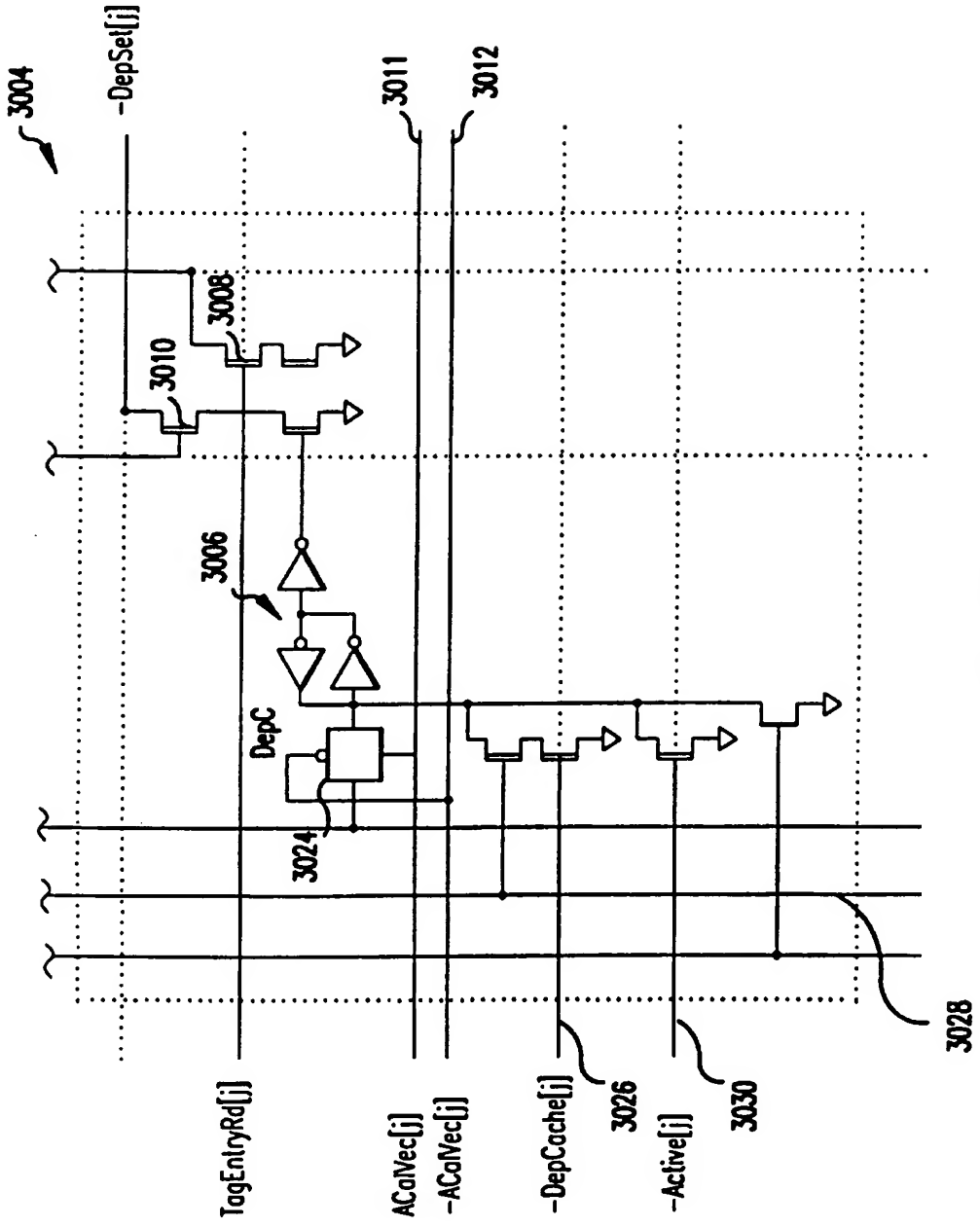

3004
-DepSel[j]
3011
3012
3008
3010
3006
DepC
3024
3028
TagEntryRd[j]
ACalVec[j]
-ACalVec[j]
-DepCache[j]
3026
-Active[j]
3030

FIG.30B

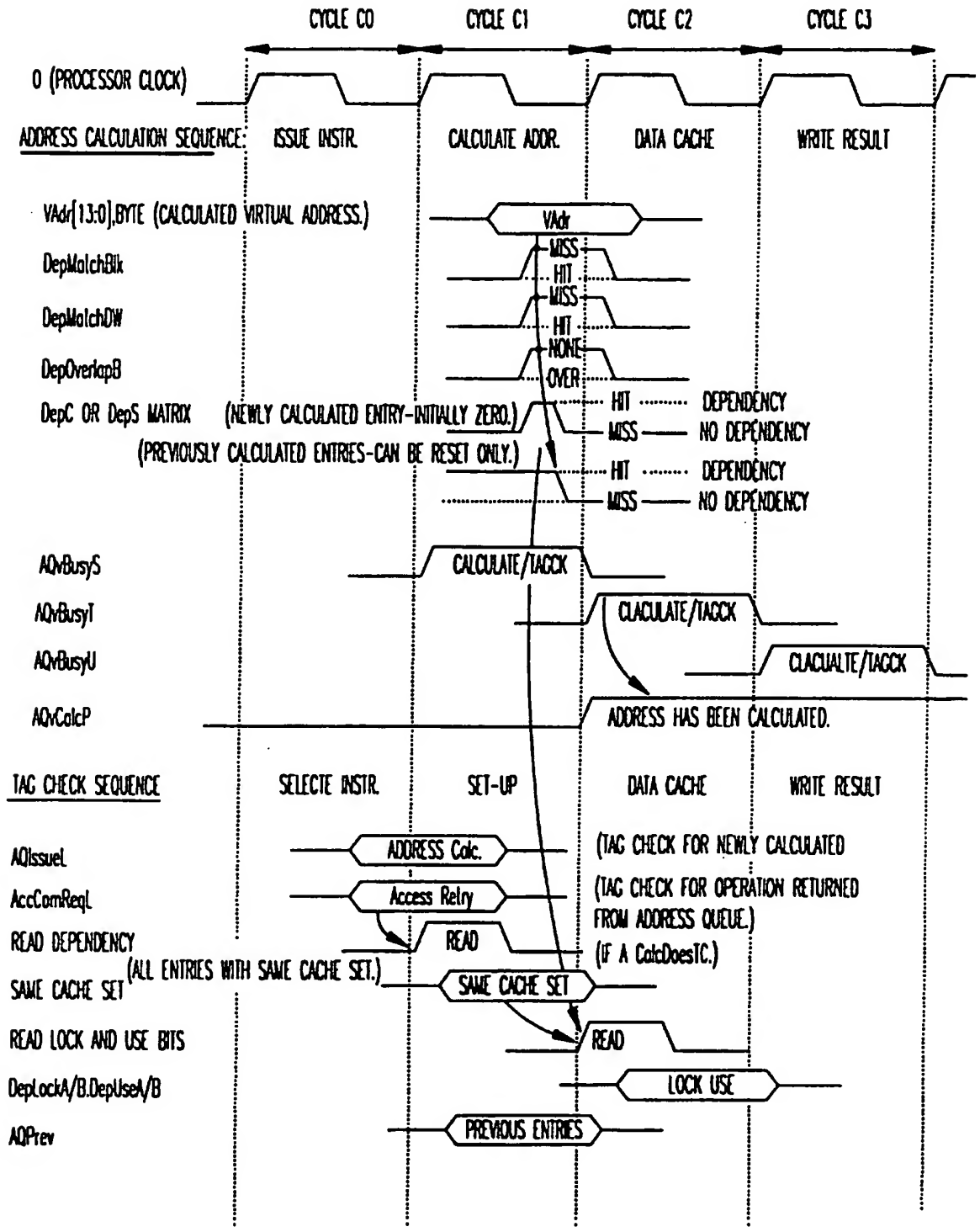

CYCLE C0
CYCLE C1
CYCLE C2
CYCLE C3
0 (PROCESSOR CLOCK)
ADDRESS CALCULATION SEQUENCE:
ISSUE INSTR.
CALCULATE ADDR.
DATA CACHE
WRITE RESULT
VAdr[13:0],BYTE (CALCULATED VIRTUAL ADDRESS.)
VAdr
DepMatchBlk
MISS
HIT
DepMatchDW
MISS
HIT
DepOverlapB
NONE
OVER
DepC OR DepS MATRIX
(NEWLY CALCULATED ENTRY-INITIALLY ZERO.)
HIT ..... DEPENDENCY
MISS —— NO DEPENDENCY
(PREVIOUSLY CALCULATED ENTRIES-CAN BE RESET ONLY.)
HIT ..... DEPENDENCY
MISS —— NO DEPENDENCY
AQvBusyS
CALCULATE/TAGCK
AQvBusyT
CLACULATE/TAGCK
AQvBusyU
CLACUALTE/TAGCK
AQvCalcP
ADDRESS HAS BEEN CALCULATED.
TAG CHECK SEQUENCE
SELECTE INSTR.
SET-UP
DATA CACHE
WRITE RESULT
AQIssueL
ADDRESS Calc.
(TAG CHECK FOR NEWLY CALCULATED
AccComReqL
Access Retry
(TAG CHECK FOR OPERATION RETURNED FROM ADDRESS QUEUE.)
READ DEPENDENCY
READ
(IF A CalcDoesTC.)
SAME CACHE SET
(ALL ENTRIES WITH SAME CACHE SET.)
SAME CACHE SET
READ LOCK AND USE BITS
READ
DepLockA/B,DepUseA/B
LOCK USE
AQPrev
PREVIOUS ENTRIES

FIG.31

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US95/13299

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : G 06 F 12/00
US CL : 395/427

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/427, 395/800

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

aps
jp abstract

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US-A-5 201 057 (UHT) ABSTRACT, 2-52 TO 3-6, 9-65 TO 9-3, 11-1 TO 11-61	1-6
A	US-A-5 467 473 (KAHLE et al.), see entire document.	1-6

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be part of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 16 JANUARY 1996	Date of mailing of the international search report 21 FEB 1996
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer FADI STEPHAN Telephone No. (703) 308-7561

Form PCT/ISA/210 (second sheet)(July 1992)*